НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Теплоенергетичний факультет
Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено:
Завідувач кафедри
_____ Олександр Коваль
«___»_____2020 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Геометричне моделювання в інформаційних системах»

спеціальності 122 «Комп'ютерні науки та інформаційні технології»

на тему: «Мікросервіс збереження профілю підрозділу університету»

Виконала:
студентка IV курсу, групи ТР-62
Поночовна Олександра Олександрівна _____

Керівник:
доцент, к.т.н.
Смаковський Денис Сергійович _____

Рецензент:
доцент, к.ф.-м.н.
Галкін Олександр Володимирович _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студентка _____

Київ – 2020

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний
Кафедра автоматизації проектування енергетичних процесів і систем
Рівень вищої освіти перший рівень
Напрям підготовки 122 Комп’ютерні науки та інформаційні технології
Спеціалізація Геометричне моделювання в інформаційних системах

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олександр Коваль
(підпис)
”___”_____2020р.

ЗАВДАННЯ
на дипломну роботу студентці

Поночовній Олександрі Олександрівні
(прізвище, ім’я, по батькові)

1. Тема роботи _"Мікросервіс збереження профілю підрозділу університету"____

керівник роботи _доцент, к.т.н. Смаковський Денис Сергійович_____
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”25” травня 2020р. № 1268-*с*

2. Строк подання студентом роботи ___16 червня 2020_____
3. Вихідні дані до роботи _Java 11, Spring Boot, Hibernate, Spring Security, Spring Data JPA, PostgreSQL, JavaScript, HTML, CSS_____

4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)___проаналізувати сучасні засоби та існуючі архітектури побудови додатку; обрати методи та засоби реалізації створюваного додатку, розробити архітектуру та структуру програмного забезпечення; обґрунтувати алгоритми роботи програми та їх реалізацію; розробити програмне забезпечення навчального призначення; зробити висновки за результатами роботи_____

5. Перелік ілюстративного матеріалу
1. Задача збереження профілю підрозділу університету. 2. Актуальність 3. Переваги мікросервісної архітектури. 4. Недоліки мікросервісної архітектури. 5. Схема побудови архітектури роботи мікросервісів. 6. Схема роботи системи підрозділів без додатку. 7. Покращена схема роботи підрозділів завдяки створеній системі. 8. Засоби реалізації системи 9. Робота користувача з інтерфейсом додатку

6. Дата видачі завдання ”_10_”___жовтня____2019_ р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	***Затвердження теми роботи***	10.10.2019	
2.	Вивчення та аналіз задачі	14.10.2019 – 23.12.2019	
3.	Розробка архітектури та загальної структури системи	02.03.2020 – 03.03.2020	
4.	Розробка структур окремих підсистем	04.03.2020 – 14.04.2020	
5.	Програмна реалізація системи	18.04.2020 – 14.05.2020	
6.	Оформлення пояснювальної записки	16.05.2020 – 07.06.2020	
7.	***Захист програмного продукту***	15.05.2020	
8.	***Передзахист***	08.06.2020	
9.	Захист	15.06.2020	

Студент _____ _____Поночовна О.О._____
(підпис) (прізвище та ініціали,)

Керівник роботи _____ _____Смаковський Д.С._____
(підпис) (прізвище та ініціали,)

АНОТАЦІЯ

Записка містить 55 сторінок, 34 рисунок, 3 додатки та 11 посилань.

Мета роботи – програмна реалізація програмного продукту, який дозволяє зберігати дані профілів підрозділів університету і мати до них доступ через цей додаток.

У результаті було розроблено систему профілю підрозділу університету. Функціональним призначенням даного сервісу є зберігання інформації про користувачів і підрозділів, з прив'язкою до організаційної структури підприємства.

Це є ефективну та безпечна система, яка дозволяє повноцінно зберігати дані доступу до баз даних та регулювати ці доступи між користувачами додатку.

Ключові слова: система регулювання даними, зберігання інформації, безпека зберігання інформаційного доступу.

ABSTRACT

The note contains 55 pages, 34 pictures, 3 appendices and 11 links.

The purpose of the work is a software implementation of a product that allows you to store data of university departments and have access to them thanks to this addition.

As a result, a university division system was developed. Functional support of this service is the storage of information about users and departments, linked to the organization of the company.

It is an efficient and secure system that allows you to save credentials to databases and manage this data received between users.

Key words: data regulation system, information storage, information data storage security.

Зміст

ВСТУП

Однією з найпоширеніших і зручних архітектур є мікросервісна архітектура. Додаток розбитий на функціональні блоки – мікросервіси, кожен з яких може маштабуватися, тим самим розподіляючи навантаження, оптимізуючи роботу всієї програми. В цьому і полягає зручність використання мікросервісної архітектури.

У даній дипломній роботі ми розглянемо реалізацію одного з мікросервісів – профілю підрозділу університету. Почнемо з функціонального призначення даного сервісу - зберігання інформації про користувачів і підрозділів, з прив'язкою до організаційної структури підприємства.

На сьогоднішній день ми не маємо ефективну систему, яка дозволяє повноцінно зберігати дані доступу до баз даних та корелювати ці доступи між користувачами додатку. Проблеми, які на разі має вирішити створення додатку:

1. Проблема збереження даних доступу до баз даних
2. Проблема регулювання доступу до баз даних між користувачами
3. Проблема надійності та безпеки зберігання інформаційного доступу до баз даних
4. Проблема зв’язку між серверами та їх створеними базами даних

Метою даної роботи є програмна реалізація програмного продукту, який дозволяє зберігати дані профілів підрозділів університету і мати до них доступ через цей додаток. Такий додаток є вдалою й ефективною заміною трудомісткої та складної схеми баз даних та доступу до них.

Для досягнення поставленої мети у роботі виконувалися такі завдання:

- проаналізувати мікросервісну архітектуру, її недоліки та переваги задля ефективної побудови архітектури нашого додатку
- обрати методи та засоби реалізації задля створення даного програмного продукту
- розробити архітектуру та структуру програмного забезпечення
- розробити програмне забезпечення навчального призначення

Структура роботи: бакалаврська робота складається зі вступу, п’ятьох розділів, висновку, списку використаних джерел і додатків. Звіт включає в себе 5 розділів.

Перший розділ описує задачу збереження підрозділів університету у мікросервісній платформі.

У другому розділі проводиться огляд мікросервісної архітектури, її переваги та недоліки.

У третьому розділі описується обґрунтування засобів реалізації системи.

Четвертий розділ описує програмну реалізацію системи, а саме архітектуру данного додатку, схему баз даних. Також у четвертому розділі детально розглядається діаграма класів, опис REST API та побудований бізнес процес.

П’ятий розділ описує програмне flow додатку.

1. ПОСТАНОВКА ЗАДАЧІ ЗБЕРЕЖЕННЯ ПРОФІЛЮ ПІДРОЗДІЛУ УНІВЕРСИТЕТУ У МІКРОСЕРВІСНІЙ ПЛАТФОРМІ

На сьогоднішній день ми не маємо ефективну систему, яка дозволяє повноцінно зберігати дані доступу до баз даних та корелювати ці доступи до них між користувачами додатку. Значному покращенню професійного рівня розробки будь–яких продуктів чи систем, а саме зберігання інформаційних даних – служить створення нової системи, що буде надійно зберігати інформаційні доступи до баз даних та регулювати доступи до них.

Метою даної дипломної роботи є створення системи, яка дозволить зберігати дані і бази даних підрозділів університету у мікросервісній платформі задля покращення контролю доступу до персональних даних. Чому це є наразі так актуально? Я вважаю, що кожна платформа, заклад кожна компанія чи команда, що розробляє продукти, або зберігає велику кількість баз даних та дані доступу до них – мають регулювати це у єдиній та цілісній системі.

Ця система є безпечною та надійною, кожен користувач має доступ тільки до відкритих йому даних до баз даних. В дипломній роботі продемонстровано створення власного програмного продукту, що являє собою систему збереження даних до баз даних, серверів та користувачів системи. Він дозволяє усім підрозділам університету мати доступ до активних серверів, баз даних, а також має користувачів – працівників підрозділів – які також мають доступ до певної інформації, яка регулюється адміністратором створених підрозділів.

Відтепер, завдяки новоствореній системі, кожен підрозділ може зареєструватися у системі та створити дані про свої сервери, їх бази даних, переглядати поточну детальну інформацію про них. Це гарантує покращення контролю доступу до даних кожного з зареєстрованих підрозділів у системі.

2. МІКРОСЕРВІСНА АРХІТЕКТУРА

Розробники усім відомої монолітної архітектури можуть роками наполегливо працювати, щоб підтримувати модульність її застосування. Також вони можуть написати комплексні автоматизовані та інтеграційні тести. Але... Чи можуть вони не уникнути проблем великої команди або декількох, яка працює над єдиною монолітною програмою? Ні. Чи можуть вони самостійно або разом вирішити проблему все більш застарілого набору технологій? На жаль, ні. Найкраще, що може зробити команда - це відстрочити неминуче. Щоб уникнути монолітного пекла, вони повинні перейти до нової архітектури: архітектури Microservice (далі архітектура мікросервісу).[1]

Якщо ви створюєте велику складну програму, вам слід подумати про використання архітектури мікросервісу. Чому саме? Які мікросервіси? На жаль, назва не допомагає, оскільки вона перебільшує розмір. Існує численні визначення архітектури мікросервісу. Адріан Коккрофт, раніше Netflix, визначає архітектуру мікросервісу як архітектуру, орієнтовану на сервіс, що складається із слабко пов'язаних елементів, що мають обмежений контекст. Це не погане визначення, але воно трохи щільне. Подивимось, чи можемо ми зробити краще. [2]

Мікросервіси - також відомі як мікросервісна архітектура, або архітектура мікросервісів, - це архітектурний стиль, який структурує програму як сукупність служб, які є:

- Нещільно з'єднані
- Незалежно розгортаються
- Організовані навколо можливостей бізнесу

У цій роботі, ми є розробниками серверного корпоративного додатку. Він повинен підтримувати безліч різних клієнтів, включаючи настільні браузери, мобільні браузери, тощо. Додаток також може виставляти API для споживання сторонніми сторонами. Він також може інтегруватися з іншими програмами через веб-сервіси. [3]

Додаток обробляє запити (HTTP-запити та повідомлення), виконуючи бізнес-логіку; має доступ до бази даних; обмінюється повідомленнями з іншими системами; і повертає відповіді HTML/JSON/XML. Також ми маємо логічні компоненти, що відповідають різним функціональним областям програми.

Розглянемо схему на рисунку 2.1., яка демонструє шаблон певного додатку, який приймає HTTP-запити від клієнтів на Api-Gateway сервіс, який є шлюзом між усіма існуючими модулями додатку. Api-Gateway сервіс надсилає необхідні запити у кожен із сервісів (service1, service2, service3)., де перевіряється наявність певних данних у database1, database2 та database3. Потім відповіді повертаються на Api-Gateway сервіс і відображуються для користувача. Тобто, сенс мікросервісів полягає у розподіленні роботи між модулями додатку, а також у інкапсуляції даних бази кожного сервісу від інших існуючих.[9]

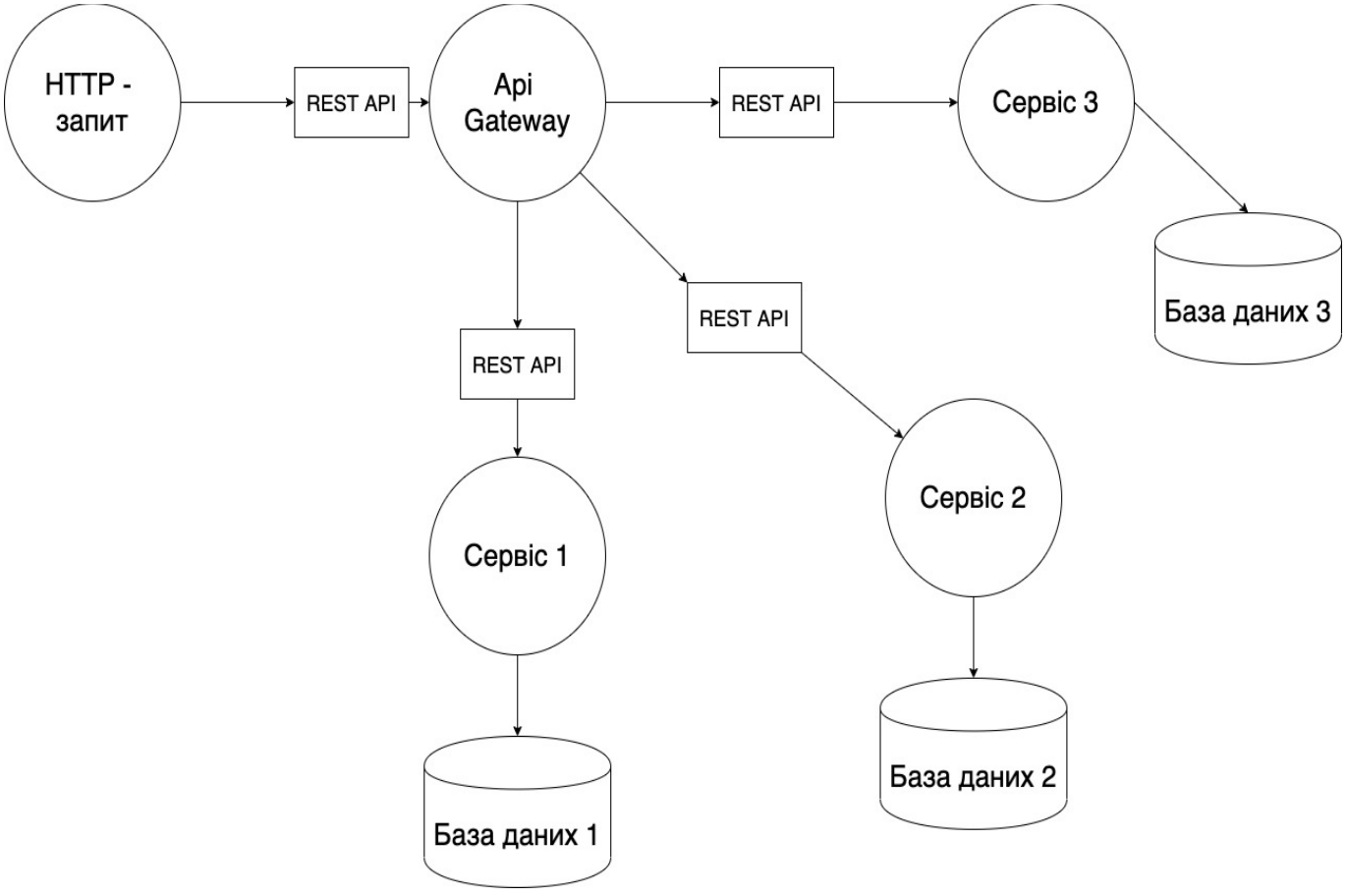

Рисунок 2.1. – Архітектура роботи мікросервісів

Модульність є важливою при розробці великих, складних додатків. Сучасний додаток, наприклад як FTGO, занадто великий, щоб його розробили окремі команди. Це також занадто складно, щоб його зрозуміла один з розробників. Програми повинні бути розбиті на модулі, які розробляються та розуміються різними командами. У монолітній програмі модулі визначаються за допомогою комбінації конструкцій мови програмування (таких як пакети Java) та побудови артефактів (наприклад, файлів Java JAR). Однак, як виявили розробники FTGO, такий підхід, як правило, не працює на практиці. [5]

Архітектура мікросервісу використовує послуги як одиницю модульності. У сервісі є API, який є непроникною межею, яку важко порушити. Ви не можете обійти API і отримати доступ до внутрішнього класу, як це можливо з пакетом Java. Як результат, набагато простіше зберегти модульність програми з часом.

Ключова характеристика архітектури мікросервісу полягає в тому, що служби слабко пов'язані та спілкуються лише через API. Досягти вільного зв'язку можливо, коли кожен сервіс має власне сховище даних. Наприклад, в інтернет-магазині у службі замовлення є база даних, що включає таблицю ЗАМОВЛЕННЯ, а у служби обслуговування клієнтів є її база даних, яка включає таблицю ЗАМОВНИКІВ. На час розробки розробники можуть змінювати схему сервісу без узгодження з розробниками, які працюють над іншими службами. Під час виконання служби послуги ізольовані один від одного - наприклад, одна служба ніколи не буде заблокована, оскільки інша служба містить блокування бази даних.

2.1. ПЕРЕВАГИ ТА НЕДОЛІКИ

Вибір саме архітектури мікросервісів має ряд переваг, які розглянути у нотованому списку нижче:

1. Архітектура мікросервісів включає розгортання великих, складних додатків, незалежних один від одного.
2. Кожен сервіс порівняно невеликий і тому його легше зрозуміти та змінити, так як кожен сервіс має якусь певну задачу та дані на вхід або вихід.

3. Запити до кожного з сервісів, як було зазначено вище, інкапсульовані від інших, тому якщо відіслати одночасно певні запити на сервіси – інформація буде швидше перевірятися та віддаватися користувачу
4. Кожен з модулей можна розгортати самостійно, незалежно від інших існуючих у додатку
5. Архітектура мікросервісів дозволяє організувати зусилля з розробки навколо декількох автономних команд. Кожна команда може розробляти, тестувати, розгортати та масштабувати свої послуги незалежно від усіх інших команд.
6. Кожен мікросервіс порівняно невеликий та виконує одну, зазначену йому, задачу
7. Простіше зрозуміти розробнику
8. Кожен з додатків запускається швидше, що робить прискореним розгортання для розробників
9. Також варто відмітити поліпшену ізоляція несправностей одного сервісу від інших. Наприклад, якщо в одному модулі є витік пам'яті, це вплине лише на цей мікросервіс.

Інші сервіси продовжуватимуть обробляти запити, без переривання процесу. Для порівняння, один невідповідний компонент монолітної архітектури може збити всю налагоджену систему. [6]

При розробці нового сервісу розробник може обрати новий стек технологій. Так само, вносячи зміни до існуючої служби, команда розробників може переписати її за допомогою нового стека технологій, без зупинення процесу всієї налагодженої системи.

Не будемо нехтувати недоліками такої архітектури, адже звичайно немає наразі ідеально працюючої системи для розробки додатків. Розглянемо ряд мінусів роботи з мікросервісною архітектурою:

1. Команда розробників повинні вирішити додаткову складність створення розподіленої системи, розділити всі задачі на мікросервіси. Це дуже важливо на етапі початку написання додатку. Адже помилки побудови можуть коштувати бізнес-стороні проекту нову перебудову додатку.

2. Реалізація запитів, що охоплюють декілька сервісів є складнішою, аніж монолітна система. Тобто, повертаючись до малюнка 1, сервіс-шлюз має правильно розподіляти відправлені дані між усіма існуючими сервісами, отримувати інформацію та відправляти користувачу необхідне.

3. Тестувати взаємодію між службами складніше

4. Реалізація запитів, що охоплюють кілька мікросервісів, вимагає ретельної координації між командами

5. Інструменти розробників орієнтовані на створення монолітних додатків і не надають явної підтримки для розробки розподілених додатків мікросервісної архітектури.[7]

6. Збільшене споживання пам'яті. Архітектура мікросервісу замінює N монолітних екземплярів додатків екземплярами служб NxM. Якщо кожна служба працює у власному JVM, що, як правило, необхідно для ізоляції екземплярів, тоді є накладні витрати в M разів більше, ніж тривалість JVM.

Зробимо висновок, що дана архітектура має досить немалу кількість як переваг, так і недоліків. Основної ідеєю використання даної архітектури є ситуація, коли ми можемо знехтувати недоліками і за рахунок необхідних для додатку переваг взяти на проект відповідальність на рахунок вибраної архітектури. Так як же зрозуміти як вибрати необхідну архітектуру на початку розробки додатку? Одним із важливих пунктів використання цього підходу є рішення, коли є сенс його використовувати.

Саме при розробці першої версії програми, досить часто не виникає проблем, які вирішує такий підхід. Більше того, використання розробленої, розподіленої архітектури сповільнить розвиток.

А як підсумок - це може стати головною проблемою для стартапів, найбільшою проблемою яких часто є те, як швидко розвивати бізнес-модель.

Однак пізніше, коли завдання команди розробників полягає в тому, як масштабувати проект і необхідним є використання функціональної декомпозицію, заплутані залежності можуть ускладнити декомпозицію вашої монолітної програми на певну кількість мікросервісів.[8]

А чи можна розкласти вже існуючий додаток на мікросервіси?

Ще одне завдання стає вирішення, як розділити систему на мікросервіси. Все ж таки існує ряд стратегій, які можуть допомогти за необхідним фінансуванням з боку бізнесу (це є необхідною частиною декомпозиції монолітної архітектури) :

1. Спочатку необхідно визначити послуги, що відповідають можливостям бізнесу, чи достатню кількість ресурсів, часу та розробників може виділити сторона бізнесу. Це питання необхідно узгоджувати з двух сторін.

2. Можно розкласти за допомогою дієслова та визначити сервіси, які відповідають за конкретні дії. Наприклад, додаток інтернет магазину може мати сервіс доставки, який відповідає лише за доставку повних замовлень та збереження у базі інформації про них, відображуючи її для кур'єрів, менеджерів, тощо. Це лише малий приклад розбиття монолітної архітектури на мікросервісну.

3. Також альтернативним варіантом є розклад за іменниками або ресурсами, визначивши службу, яка відповідає за всі операції з об'єктами / ресурсами певного типу. [9]

Наприклад, ми маємо додаток соціальної мережі. Отже, один із сервісів може бути сервісов акаунтів, який відповідає за управління обліковими записами користувачів.

4. В ідеалі кожна служба повинна мати лише невеликий набір обов'язків. Боб Мартін, у своїй книзі «Чиста архітектура. Мистецтво розробки», описує можливі варіанти розробки класів за принципом єдиної відповідальності (SRP).

Також принцип єдиної відповідальності є одним із п'яти принципів SOLID, ці принципи використовуються для дизайну та розробки таких програмних систем, які, з великою ймовірністю, зможуть тривалий час розвиватися, розширятися і підтримуватися.

2.2. АНАЛІЗ ІСНУЮЧИХ СИСТЕМ

На сьогоднішній день ми не маємо ефективну систему, яка дозволяє повноцінно зберігати дані доступу до баз даних та корелювати ці доступи між користувачами додатку. Розглянемо умовну схему на рисунку 2.2. – підрозділ має користувачів, які мають дані для доступу до баз даних.

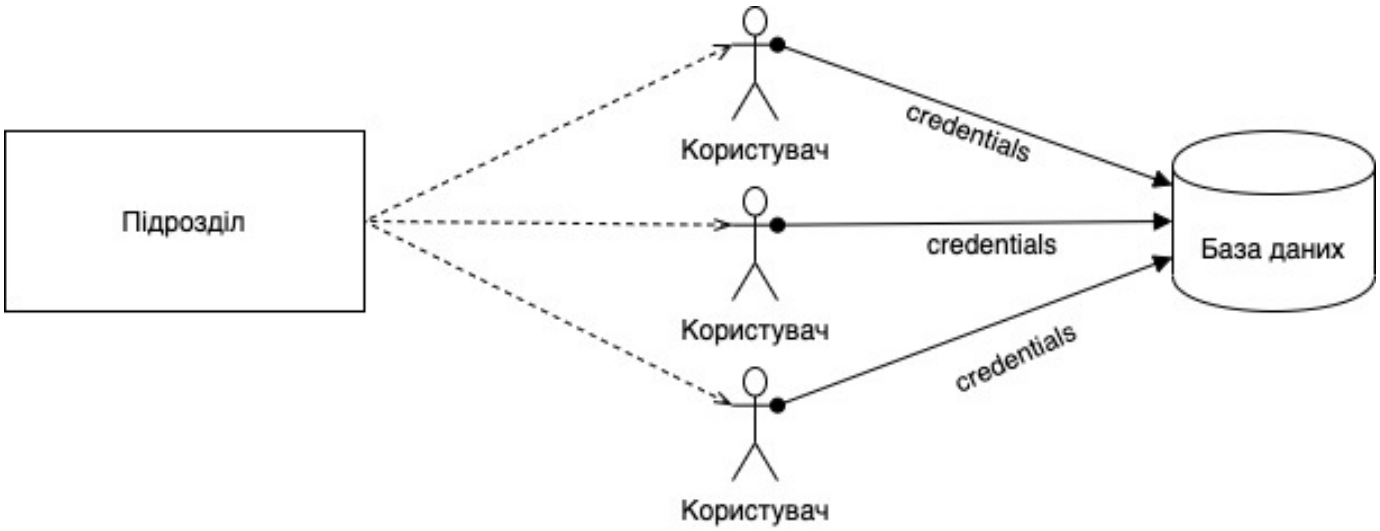

Рисунок 2.2. – Схема роботи без створеного додатку

Перейдемо до іншої умовної схеми, що зображена на рисунку 2.3. – підрозділ має користувачів, які мають доступ до додатку, який регулює бази даних підрозділу та віддає існуючі доступи до баз даних.

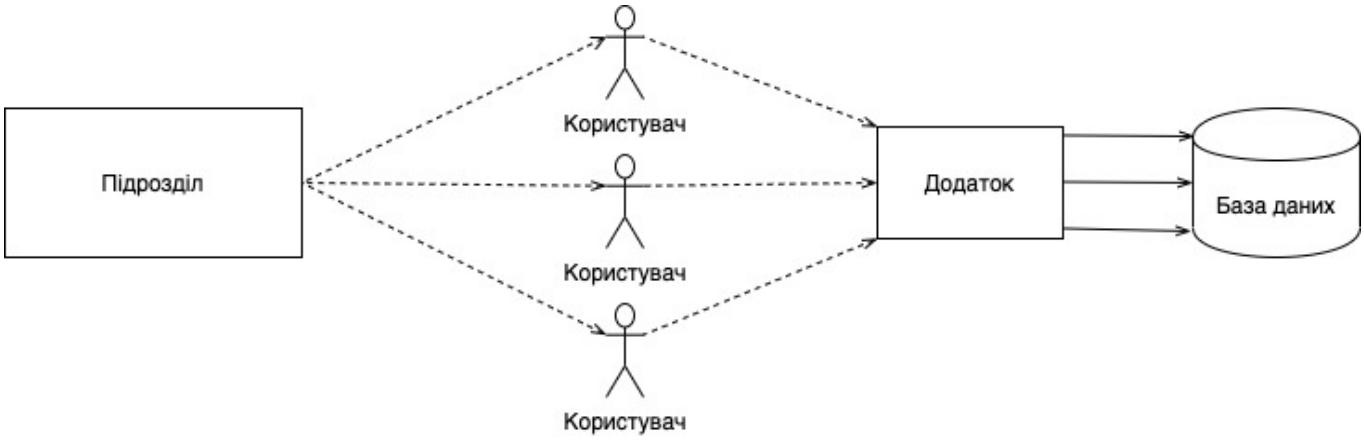

Рисунок 2.3. – Схема роботи з створеним додатком

Це значно покращує контроль доступу до баз даних. Це і є основною метою створення додатку. Він дозволяє усім підрозділам університету мати доступ до активних серверів, баз даних, а також має користувачів – працівників підрозділів –

які також мають доступ до певної інформації, яка регулюється адміністратором створених підрозділів.

Відтепер, завдяки новоствореній системі, кожен підрозділ може зареєструватися у системі та створити дані про свої сервери, їх бази даних, переглядати поточну детальну інформацію про них. Це гарантує покращення контролю доступу до даних кожного з зареєстрованих підрозділів у системі.

3. ОБҐРУНТУВАННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ СИСТЕМИ

При розробці програмних продуктів одним із найважливіших завдань є вибір засобів, які полегшують роботу програміста, а саме – надають всі інструменти для необхідної реалізації поставленого завдання, і дають змогу отримувати результат, який задовольняє користувача повністю.

3.1. Технології для взаємодії з клієнтською частиною

Клієнтська частина продукту:

1. Мова програмування JavaScript;
2. Мова розмітки веб-сторінок HTML;
3. Мова розмітки CSS для розробки графічного інтерфейсу користувача;

Клієнтську частину мого додатку можно умовно поділити на 3 основні частини:

Першою частиною є **HTML** – це мова розмітки, яка використовується для структурування веб-вмісту, а саме, виділення різних заголовків, абзаців, тощо.

Другою частиною є **CSS** – це мова стилів, яка використовується для застосування стилів до коду HTML, наприклад, встановлення кольорів тексту та шрифтів і так далі.

Третьою і найголовнішою частиною побудови клієнтської частини додатку є **JavaScript** – це мова, яка використовується для створення динамічних сторінок, а також, за допомогою цієї мови - ми можемо керувати відображенням мультимедіа, анімаціями, тощо.

Ми будемо використовувати HTML5 – сама нова версія, з якою браузери, такі як Firefox, Chrome, Explorer, тощо, знають, як відображати певну веб-сторінку, куди саме треба розмістити зображення або текст.

Крім HTML5, звичайно ж, існують й інші мови, але все ж таки основна структура будь-якої веб-сторінки структурується завдяки мові HTML5.

Веб-браузери застосовують правила документа CSS до документа, щоб вплинути на те, як вони відображаються.

Правило CSS формується з:

1. Набір властивостей, які мають значення, як відображається вміст HTML.
2. Селектор, який вибирає елемент ,до якого потрібно застосувати оновлені значення властивостей.
3. Набір правил CSS, що містяться в таблиці стилів, визначає, як має виглядати веб-сторінка.

Як працює CSS? Коли браузер відображає документ, він повинен поєднувати вміст документа з його інформацією про стиль.

Він обробляє документ у два етапи:

1. Браузер перетворює HTML і CSS в DOM (Document Object Model). DOM являє собою документ у пам'яті комп'ютера.
2. Браузер відображає вміст DOM

JavaScript змінює "динамічний вміст веб-сайту" на "речі, які переміщуються, змінюються на екрані, не вимагаючи ручного перезавантаження веб-сторінки".
До "динамічного вмісту сайту "можна віднести такі функції, як анімаційна графіка, слайд-шоу фотографій, пропозиції авто заповнення тексту та інтерактивні форми.

3.2. Технології для взаємодії з сервером

Серверна частина продукту:

1. Мова програмування Java;
2. Інструмент для збірки Java проекту Gradle;
3. Фреймворк Spring Security;
4. Фреймворк Spring Boot;

База даних продукту:

1. Модуль для роботи з базою даних PosgtreSQL;

Ще одним важливим завданням є вибір операційної системи, на якій буде розроблятися програмний продукт. Було обрано середовище MAC ОС, оскільки дана операційна система є сучасною, найпоширенішою, багатофункціональною, добре захищеною, зручною та надійною. Також обрана ОС сумісна з іншими операційними системами та відповідає вимогам сучасних програмних засобів.

Оскільки створений програмний продукт не залежить від операційної системи, то може бути використаний як комп'ютер з ОС Windows, так і з UNIX.

Можливість перегляду результатів роботи лише браузер. Виходячи з того, що даний програмний продукт поділяється на серверну, клієнтську частини та базу даних, то зупинимося на засобах, які використовувались і спробуємо проаналізувати їх переваги та недоліки.

Платформа **Java –** це лише програмне забезпечення, яке значно відрізняється від традиційних платформ, таких як Windows, Mac, Linux або Solaris.

Програми Java проходять через віртуальну машину Java, яка перетворює байт-код у нативний код, завдяки чому програма запускає будь-який пристрій! Це означає, що для запуску коду Java вам не потрібні окремі компілятори. [10]

Ось чому Java також називається платформою та високо-інтелектуальною мовою програмування.

Розглянемо переваги вибору саме Java задля побудови нашого додатку:

1. Незалежність платформи

Однією з головних причин Java настільки популярною, як було сказано вище, це її незалежність від платформи, а це дає нам знати про те, що програми Java можна запускати на різних типах комп'ютерів і операційних системах. Програма Java працює на будь-якому комп'ютері із встановленим середовищем виконання Java, також відомим як JRE. JRE доступний майже для всіх типів комп'ютерів – ПК, на яких працює Unix або Linux, ОС Windows, комп'ютерів Mac. і навіть на звичайних телефонах.

2. Орієнтація на об'єкт

Java по своїй суті є об'єктно-орієнтованим, що означає, що програми Java складаються з елементів програмування, званих об'єктами. Простіше кажучи, ми створюємо об’єкти навколо нас, для розуміння об’єктом є програмуючий об'єкт, який представляє або якийсь об'єкт реального світу, або це може бути абстрактне поняття.

3. API Java

Java має велику бібліотеку класів, які надають багато потрібних речей і базових функцій, без яких не може обійтися більшість програм Java. Мова Java містить лише 50 ключових слів, але Java API має кілька тисяч класів – з десятками тисяч методів, які можна використовувати у своїх програмах. Саме тому, програміст, який обрав Java – вчить не саму мову програмування, а її API.

Gradle – це інструмент автоматизації побудови проектів Java. Якщо вам доводилося мати справу зі створенням Java-проекту з залежностями або спеціальними вимогами для збирання, ви, мабуть, переживали розчарування, які Gradle прагне усунути. Він є дуже стабільним і багатофункціональним, забезпечуючи численні плагіни, які можуть зробити що-небудь від генерації PDF-версій документації вашого проекту до створення списку останніх змін з вашого SCM. І все, що потрібно, щоб додати цю функціональність – це невелика кількість додаткового XML або додатковий параметр командного рядка. Є багато залежностей? Без проблем. Gradle підключається до віддалених сховищ (або ви можете налаштувати власні місцеві репозитарії) і автоматично завантажує всі залежності. [13]

Spring Boot Framework – це середовище на основі Java з відкритим вихідним кодом, яке використовується для створення мікросервісів.

Spring Boot пропонує наступні переваги для своїх розробників:

- Легко зрозуміти і розробляти додатки
- Збільшує продуктивність
- Скорочує час розробки

Spring Boot розроблений для наступних цілей:

- Щоб уникнути складної конфігурації XML в Spring
- Розробляти готові програми Spring простіше
- Скоротити час розробки і запустити додаток самостійно

Ми обрали Spring Boot через функції і переваги, які він пропонує:

1. Він надає гнучкий спосіб настройки Java Beans, конфігурацій XML і транзакцій бази даних.
2. Він забезпечує потужну пакетну обробку і управляє кінцевими точками REST.
3. В Spring Boot все налаштовується автоматично; ручні настройки не потрібні.
4. Полегшує управління залежностями
5. Включає вбудований контейнер сервлетів
6. Він надає гнучкий спосіб настройки Java Beans, конфігурацій XML і транзакцій бази даних.
7. Він забезпечує потужну пакетну обробку і управляє кінцевими точками REST. [7]

Подумайте про Spring Boot як набір, який надає попередньо налаштований набір бібліотек з усіма їх залежностями, виконаними для зменшення конфігурації котла. Це найшвидший спосіб створити готовий до виробництва додаток, який працює. Що робить Spring Boot, це те, що він дотримується конвенції щодо конфігурації та пропонує вибрати відповідний набір бібліотек.

Це робиться з невеликим втручанням програміста. Spring Boot імпортує необхідні бібліотеки, їх залежності та всі залежності, яких умовно вимагає тип проекту, наприклад сервер додатків тощо.[14]

Упаковка та розгортання програми, готової до виробництва, також не викликає клопоту. Можна майже забути, що додаток насправді працює на вбудованому сервері додатків, коли створена банка проектів. Додаток працює як окремий додаток.

Управління залежностями – складне завдання для великих проектів. Spring Boot вирішує цю проблему, надаючи набір залежностей для зручності розробників.

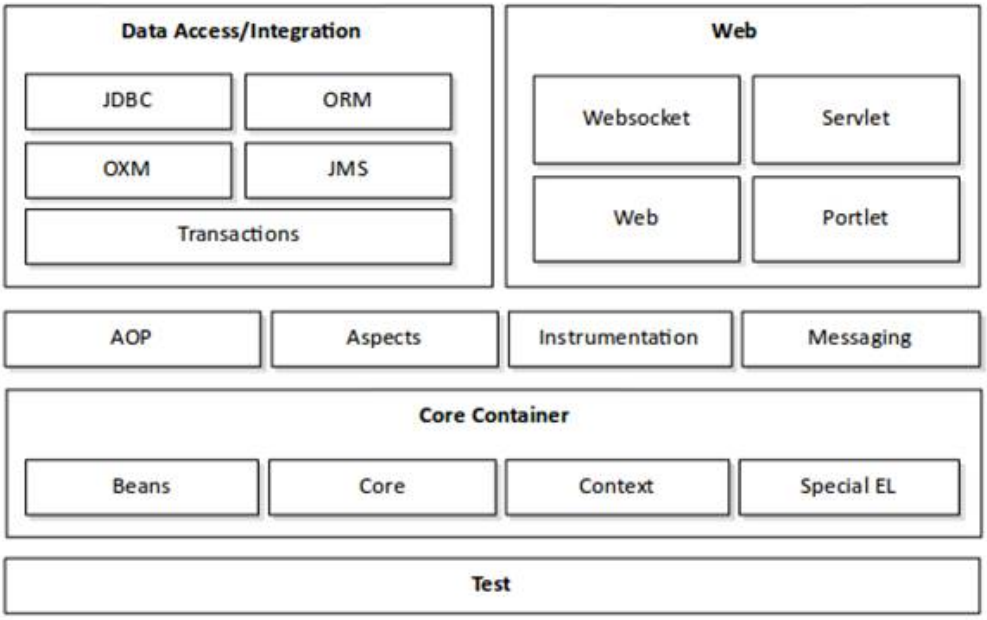

Рисунок 3.1. – Spring framework

Розглянемо для початку Core контейнер та його складові задля повного розуміння роботи даного фреймворку.

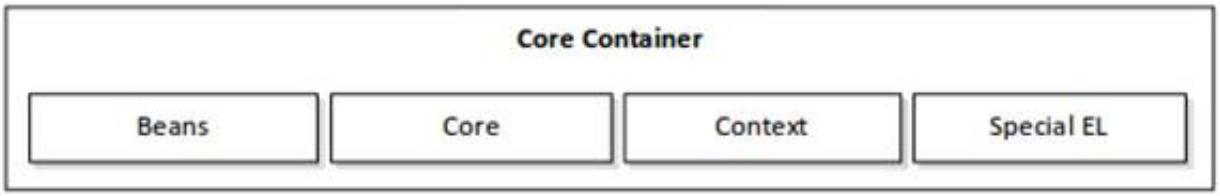

Рисунок 3.2. – Core контейнер та його складові

Core контейнер – Spring-Core та Spring-Beans модулі складають основну частину Spring Framework та включають функції IoC (інверсія управління) та функції введення залежностей.

Ідея IoC стверджує, що замість програми, що викликає методи, фреймворк буде робити цей виклик.

Звідси назва Inversion of Control, або IoC, коротше. Однак IoC – це більш широкий термін, а функція введення залежностей (DI) – форма його реалізації. DI усуває щільне з'єднання між компонентами програмного забезпечення шляхом введення властивостей об'єкта через конструктор та інжектор геттера / сетера. [14]

Spring-Context модуль побудований на модулях Spring-core та Spring-ApplicationContext - це основний інтерфейс, похідний від цього модуля. Модуль підтримки Spring-Context забезпечує можливість інтеграції багатьох бібліотек, таких як JasperReport для звітування, Quartz для планування, JavaMail для розсилки та EhCache для кешування.

Модуль Spring-Aop забезпечує реалізацію для орієнтовано-аспектного програмування. Основним моментом аспектно-орієнтованого програмування (AOP) є відокремлення наскрізних проблем від ділової логіки, хоча вони беруть участь у загальній роботі програми, вони все ще є окремими концептуально. Дуже поширеним прикладом є процес реєстрації, який може перекрити стурбованість бізнес-логіки, оскільки він не має явної потреби в цьому процесі, але допомагає реєструвати події, які можуть повернутись до певної контрольної точки або налагодити програму в довгостроковій перспективі. [14]

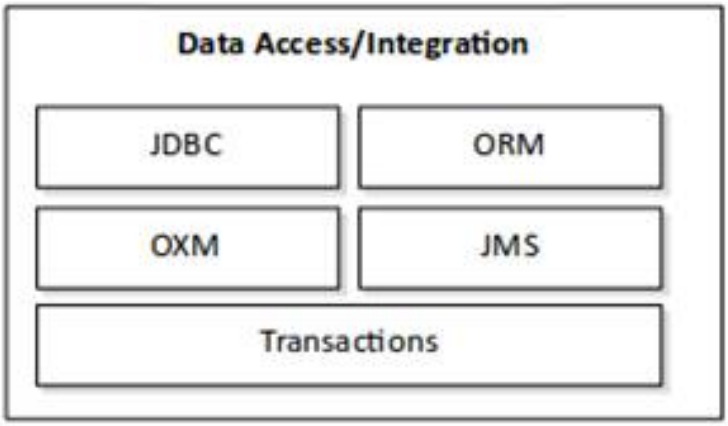

Рисунок 3.3. – Data Access Integration

Рівень доступу до даних / інтеграція містить підтримку JDBC, ORM, OXM, JMS та транзакцій.

Модуль **Spring JDBC** забезпечує абстракцію JDBC.

Модуль **Spring TX** підтримує програмні та декларативні транзакції для всіх POJO.

Модуль **Spring ORM** забезпечує підтримку популярних API ORM, таких як JPA, JDO, Hibernate тощо.

Модель **Spring OXM** призначена для реалізації об'єктних / XML-карт, таких як JAXB, XStream, Castor та XMLBeans.

Модуль **Spring JMS** – це абстрагування служби обміну повідомленнями Java (JMS). Він надає підтримку для створення та споживання повідомлень. З Spring 4.1 цей модуль інтегрований з модулем Spring Messaging.

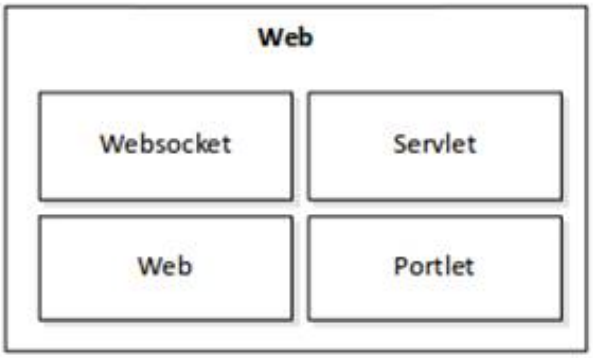

Рисунок 3.4. – Веб-шар Spring Framework

Веб-шар складається з чотирьох модулів: Spring Web, Spring Web MVC, Spring Websocket та Spring Web MVC portlet.

Модуль **Spring Web** є основою інтеграції веб-функцій, таких як функція завантаження файлів із декількома частинами та ініціалізація контейнера IoC із слухачем Servlet. Він також містить клієнт HTTP та пов'язані з ним частини для підтримки віддалення.

Модуль **Spring Web MVC** забезпечує підтримку послуг MVC та REST для веб-додатків.

Модуль **Spring Websocket** забезпечує підтримку протоколу Web socket, визначеного RFC 6455. Це забезпечує можливість веб-додатків підтримувати повнодуплексний зв'язок між клієнтом і сервером.

Spring Web MVC portlet забезпечує підтримку впровадження MVC для середовища Portlet, подібного до реалізації сервлетів у модулі **Spring Web MVC**.

Модуль пружинного тестування забезпечує підтримку тестування модулів, тестування інтеграції спільно з **JUnit або TestNG.**

Spring Security Framework – це фреймворк, який сфокусований на забезпечення як аутентифікації, так і авторизації в Java-додатках.

Як і всі Spring проекти, справжня сила Spring Security в тому, що він може бути легко доповнений за потрібне функціоналом. Можливості:

1. Комплексна і розширюється підтримка як аутентифікації, так і авторизації
2. Захист від атак типу фіксація сесії, клікджекінг, міжсайтовий підробка запиту і ін.
3. Інтеграція з Servlet API
4. Інтеграція з Spring Web MVC при необхідності

Spring Security - це потужний і налаштований шаблон аутентифікації та контролю доступу. Це фактичний стандарт для забезпечення аутентифікації додатків на базі Spring.

Spring Security - це бібліотека, яка зосереджена на наданні як аутентифікації, так і авторизації додаткам Java. Як і всі проекти Spring, реальна сила Spring Security полягає в тому, наскільки легко його можна розширити для задоволення спеціальних вимог.

Особливості:

1. Повна та розширена підтримка як для аутентифікації, так і для авторизації
2. Захист від атак, таких як підробка веб-сайтів, тощо
3. Інтеграція API сервлетів
4. Необов’язкова інтеграція з Spring Web MVC

Spring Security - це структура, яка дозволяє програмісту накладати обмеження на безпеку веб-додатків, заснованих на Spring-Framework. Коротше кажучи, це бібліотека, яку можна використовувати для налаштування відповідно до потреб програміста.

Оскільки він є членом тієї ж родини Spring, він ідеально поєднується з MVC Spring Web. Можливо, найбільшою перевагою цього фрейворку є те, що він є потужним, але гарно налаштованим у своїй реалізації. Хоча це відповідає умовам Spring щодо конфігурації, програмісти можуть вибирати положення за замовчуванням або налаштовувати його відповідно до своїх потреб.

PostgreSQL – не просто реляційна, а об'єктно-реляційна СУБД. Це дає деякі переваги над іншими SQL базами даних з відкритим вихідним кодом, такими як MySQL, MariaDB і Firebird. Фундаментальна характеристика об'єктно-реляційної бази даних - це підтримка об'єктів і їх поведінки, включаючи типи даних, функції, операції, домени і індекси. Це робить PostgreSQL неймовірно гнучким і надійним. Серед іншого, він вміє створювати, зберігати та видавати складні структури даних.

Крім числових, з плаваючою точкою, текстових, булевих і інших очікуваних типів даних (а також безлічі їх варіацій), PostgreSQL може похвалитися підтримкою UUID, грошового, геометричного, бінарного типів, мережевих адрес, бітових рядків, текстового пошуку, xml, json , масивів, композитних типів і діапазонів, а також деяких внутрішніх типів для ідентифікації об'єктів. Справедливості заради варто сказати, що MySQL, MariaDB і Firebird теж мають деякі з цих типів даних, але тільки PostgreSQL підтримує їх всі. PostgreSQL забезпечує розширену ємність даних і заслужив довіру дбайливим ставленням до цілісності даних. [14]

4. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

4.1. АРХІТЕКТУРА ДОДАТКУ

Модель (Model / Entity) у патерні Spring MVC (что є основою Spring Boot Framework) - це шар, який представляє логіку Вашого проекту, абстраговані від будь-яких деталей UI, завдань перетворення даних і т.д. Так само модель відома як Java POJO (Plain Old Java Object) - це простий Java об'єкт.

Контролер (Controller) – обробляє запит користувача зі сторони UI, створює відповідну Model (з переданих користувачем даних) передає її на backend задля обробких цих даних. В результаті користувач отримує відображення данних, які передав backend. Отже, підсумуємо, що контроллер поєднує разом бізнес логіку і логіку оброблення вхідних даних, передаючи їх відповідно, а також він відповідає за формування відповіді користувачу.

Репозиторій (Repository) – це набір інтерфейсів які використовують JPA Entity для взаємодії з нею. А також – це директорія, яка містить файли класу, які відмічені аннотацією @Repository.

Так, наприклад, інтерфейс CrudRepository забезпечує основні операції з пошуку, збереження, видалення даних (CRUD операції). Так само можна прямо розширити базовий інтерфейс для своєї сутності, доповнити його своїми методами запитів і виконувати операції.

Сервіс (Service) – це директорія, яка містить файли класу, які відмічені аннотацією @Service. Основною метою створення сервісів є відокремлення бізнес логіки від контроллерів та репозиторіїв.

Application properties – це директорія або пакет, де знаходяться файли налаштувань, а загалом файли підключення до баз даних, файли налаштування логування, а також необхідні значення, що зчитуються компілятором задля подальшої необхідної обробки та використання у додатку.

4.2. КОНЦЕПТУАЛЬНА МОДЕЛЬ БАЗ ДАНИХ

Зберігання та отримання доступу до даних підрозділів університету надає створена схема баз даних нашого додатку, яка складається з чотирьох взаємопов’язаних таблиць реляційної бази даних: Сервери (“servers”), Підрозділи (“tenant”), Таблиці Баз Даних Користувачів (“db_pertenant”), Користувачі (“users”) користувачі підрозділів та проміжна таблиця (“tenant_user”), що зв’язує таблиці Користувачі (“users”) та Підрозділи(“tenant”) зв’язком багато до багатьох.

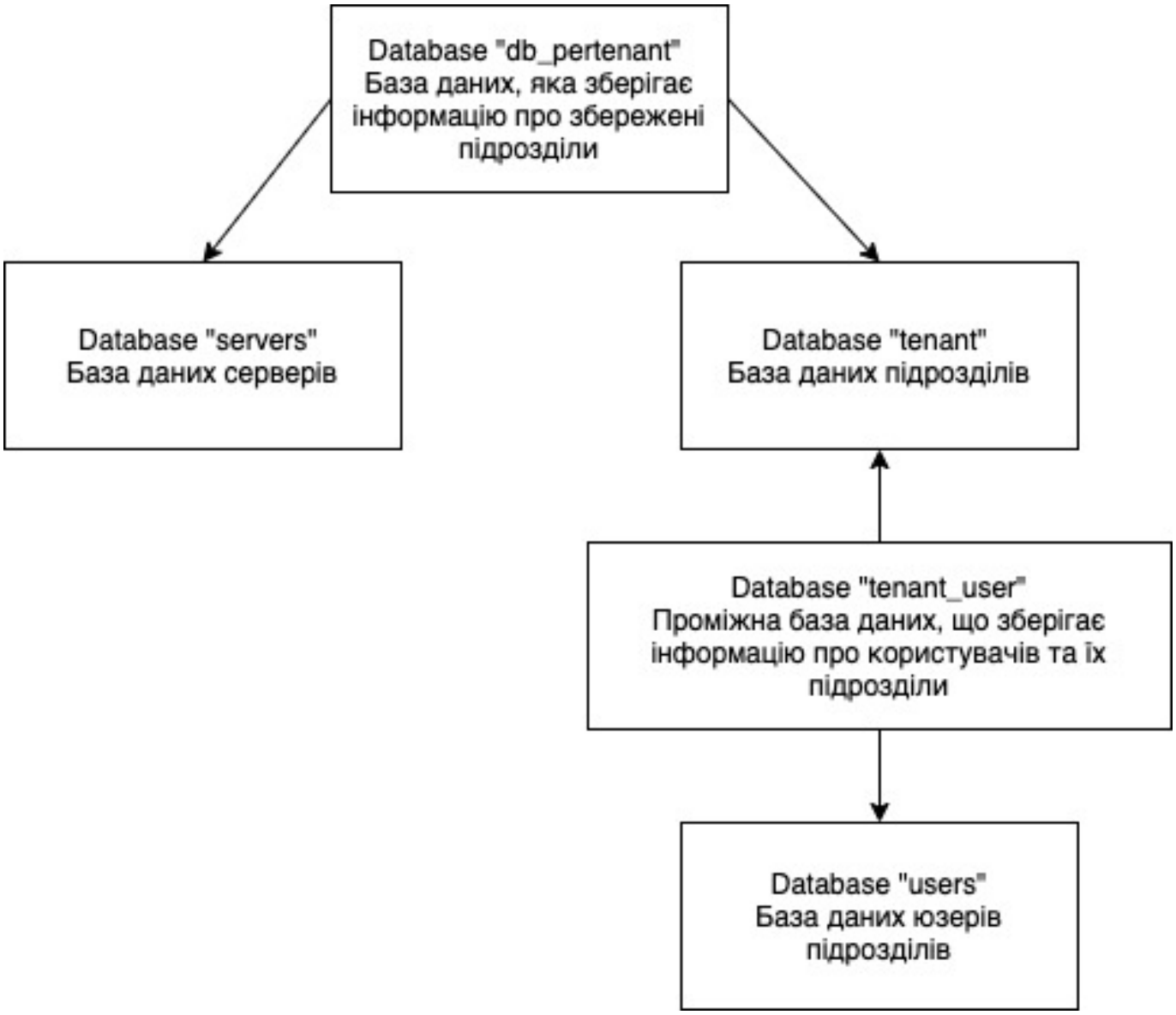

Рисунок 4.1. – Концептуальна модель бази даних

4.3. ОПИС ТАБЛИЦЬ БАЗ ДАНИХ

База даних нашого додатку реалізована за допомогою реляціонної системи управління базами даних – PostgreSQL.

У таблицях нижче 4.1 — 4.4. показана детальна інформація про існуючі бази даних у нашому додатку (ім’я поля, тип і розмір поля та опис поля).

Таблиця 4.1. – Структура таблиці “Сервери”

Ім’я поля, що задане	Тип і розмір поля	Опис поля згідно додатку
id	bigint	Первинний ключ
server_id	varchar(255)	Зовнішній ключ, який дозволяє пов’язати дані поточної таблиці з іншими
url	varchar(255)	Посилання на базу даних підрозділу університету
login	varchar(255)	Логін для доступу до бази даних підрозділу університету
password	varchar(255)	Пароль для доступу до бази даних підрозділу університету
counts	integer	Кількість таблиць, створених у поточній базі даних
active	boolean	Поле, яке визначає активним є даний сервер чи ні

Таблиця 4.2. – Структура таблиці “Підрозділи”

Ім’я поля, що задане	Тип і розмір поля	Опис поля згідно додатку
id	bigint	Первинний ключ
tenant_id	varchar(255)	Зовнішній ключ, який дозволяє пов’язати дані поточної таблиці
tenant_alias	varchar(255)	Ім’я підрозділу
description	varchar(255)	Опис підрозділу
active	boolean	Поле, яке визначає активним є даний підрозділ чи ні

Таблиця 4.3. – Структура таблиці “Дані Баз Користувачів”

Ім’я поля, що задане	Тип і розмір поля	Опис поля згідно додатку
id	bigint	Первинний ключ
tenant_id	varchar(255)	Зовнішній ключ, зв’язок з таблицею «Підрозділи»
service_name	varchar(255)	Ім’я сервісу, що відповідає за з’єднання з поточною базою даних
db_name	varchar(255)	Ім’я таблиці бази даних
server_id	varchar(255)	Зовнішній ключ, зв’язок з таблицею «Сервери»
db_login	varchar(255)	Логін для доступу до бази даних
db_password	varchar(255)	Пароль для доступу до бази даних
active	boolean	Поле, яке визначає активним є даний сервер чи ні

Таблиця 4.3. – Структура таблиці “Користувачі”

Ім’я поля, що задане	Тип і розмір поля	Опис поля згідно додатку
id	bigint	Первинний ключ
user_id	varchar(255)	Зовнішній ключ, зв’язок з таблицею «Підрозділи»
tenant_id	varchar(255)	Зовнішній ключ, який дозволяє пов’язати дані поточної таблиці з іншими
first_name	varchar(255)	Ім’я користувача
last_name	varchar(255)	Призвіще користувача
email	varchar(255)	Пошта користувача
login	varchar(255)	Логін користувача

password	varchar(255)	Пароль користувача
active	boolean	Поле, яке визначає активним є даний користувач чи ні

Таблиця 4.4. – Структура таблиці “Користувачі”

Ім’я поля, що задане	Тип і розмір поля	Опис поля згідно додатку
id	bigint	Первинний ключ
user_id	varchar(255)	Зовнішній ключ, зв’язок з таблицею «Підрозділи»
tenant_id	varchar(255)	Зовнішній ключ, який дозволяє пов’язати дані поточної таблиці з іншими

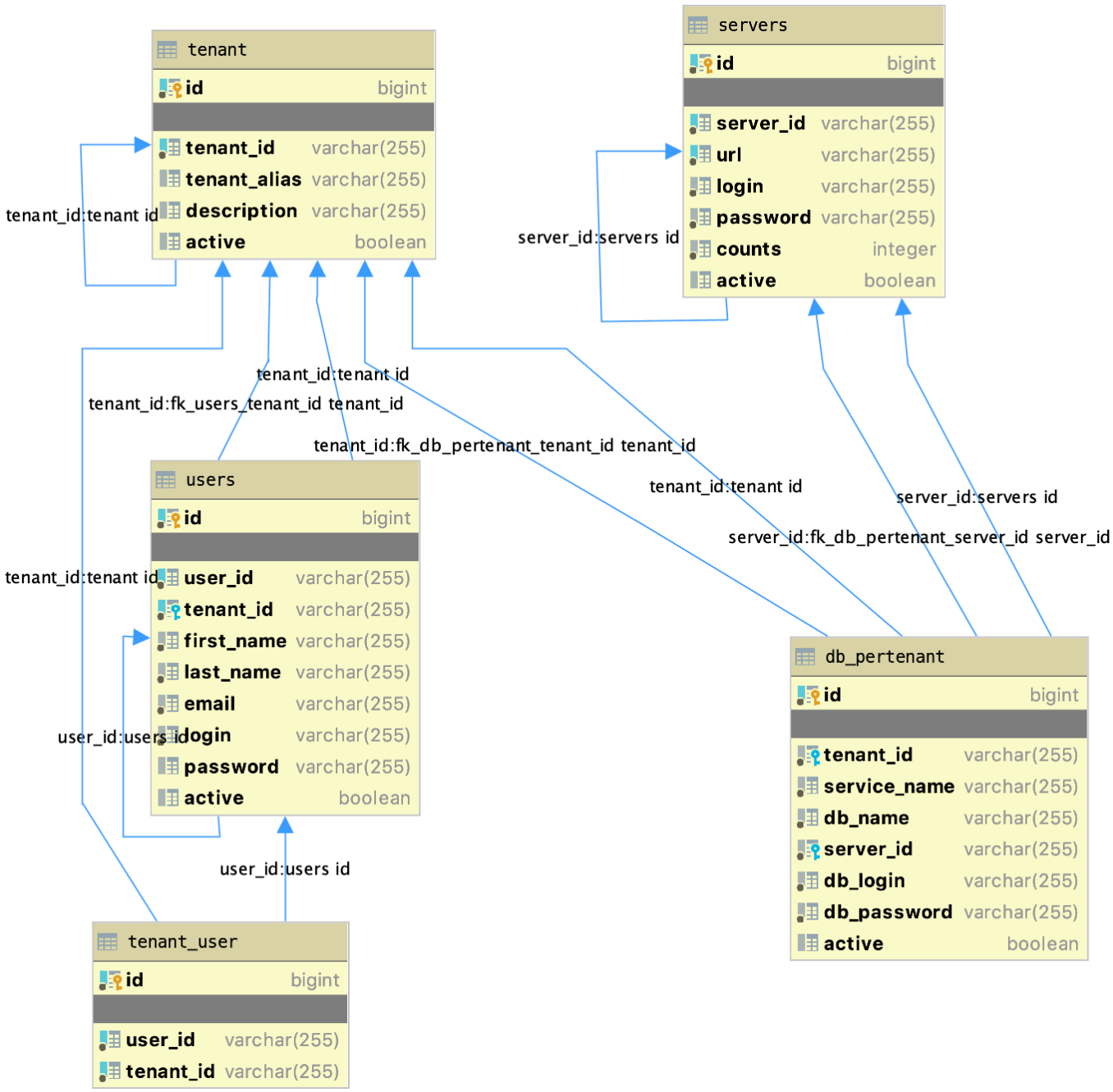

Рисунок 4.2. – Повна схема баз даних

4.4. ДІАГРАМА КЛАССІВ ДОДАТКУ

UML (Unified Modeling Language) – це система позначень, яку використовують для об'єктно-орієнтованого аналізу. Його наприклад також можливо застосувати для візуалізації.

Діаграма – це графічне представлення, частіше всього зображеного у вигляді зв'язного графа.

Діаграма класів (class diagram) існує для подання статичної структури. Властивість відображати, зокрема, всілякі зв'язки між окремими сутностями предметної області – це перша складова діаграми класів,

Важливо відмітити, що діаграма класів не має показувати динамічну поведінку об'єктів, які зображених на ній.

На діаграмах класів показуються пакети, класи, інтерфейси і відносини між ними. [9]

Діаграма класів має можливість ще показувати залежність інтерфейсів один від одного, описувати структуру пакетів, а також окремих об'єктів.

У разі якщо діаграма класів вважається частиною деякого пакета, то її компоненти повинні відповідати складовим цього пакета, охоплюючи ймовірні посилання на складові з інших пакетів.

Графічно клас відображається у вигляді прямокутника, прямокутник може розділятися горизонтальними лініями на інші прямокутники. У них вказується ім'я, атрибути властивості і методи. Також задля побудови були використані відношення асоціації, залежності, узагальнення, композиції та агрегації

На рисунках 4.3. – 4.7. нижче зображені діаграми класів основних пакетів сервісу нашого додатку.

Почнемо розглядати кожен з поданих малюнків з діаграми класів, яка описує структура пакетів сервісу нашого додатку.

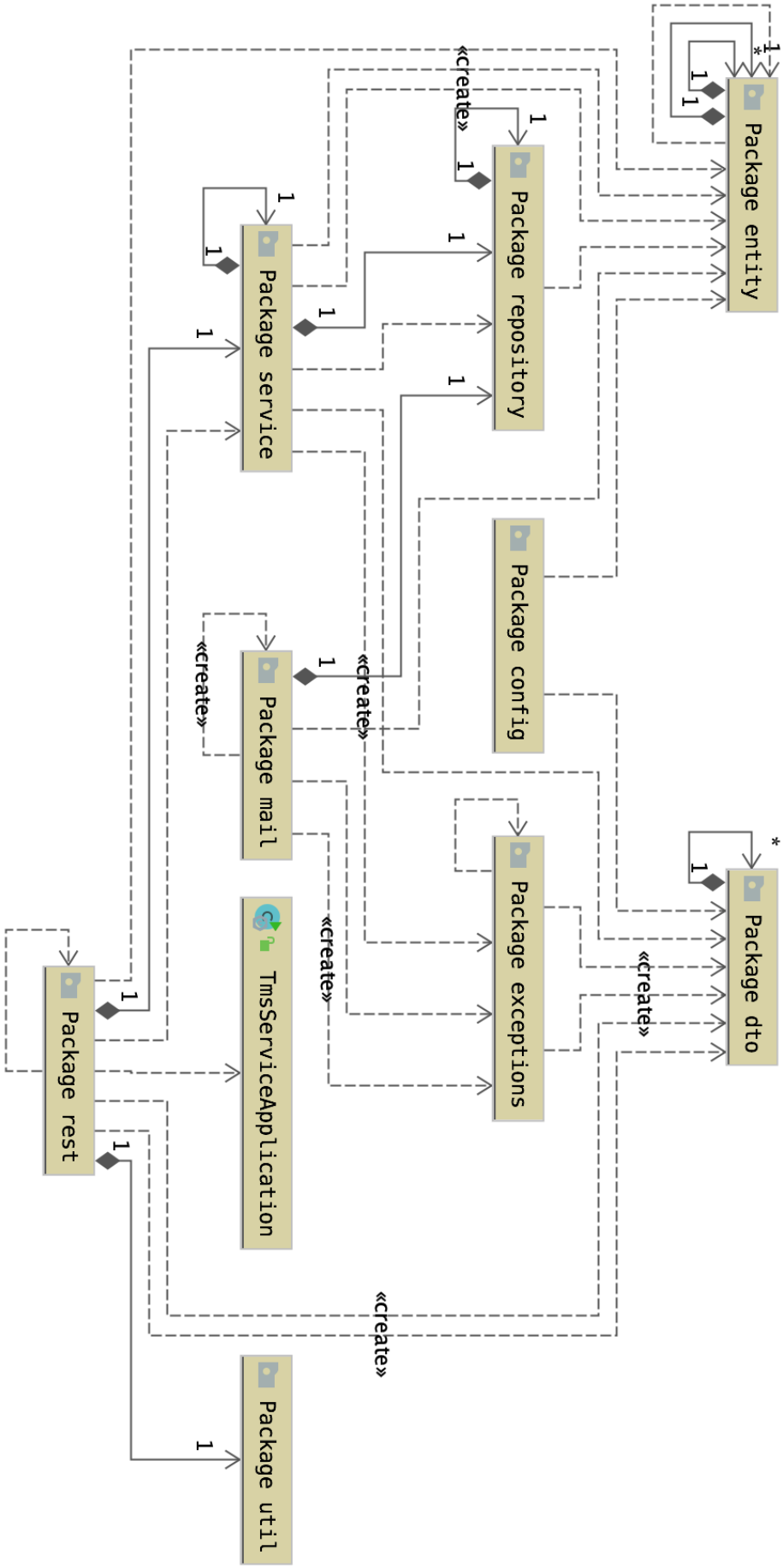

Рисунок 4.3. – Діаграма класів пакетів сервісу додатку

Пакет exceptions – включає в себе створені виключення для відображення їх повідомлень користувачу.

Пакет repository – включає в себе інтерфейси для роботи з базою даних, а саме дозволяє оперувати отриманими моделями з контролера – зберігати їх, обновлювати інформацію, видаляти, або віддавати певні дані.

Пакет service – включає в себе класи, які є проміжним шаром між контролерами і репозиторіями – містить бізнес логіку.

Пакет config – включає в себе налаштування конвертації DTO (Data Transfer Object), отриманої від користувача у Entity для зберігання в базу даних.

Пакет entity – включає в себе класи Entity, дані яких зберігаються у БД

Пакет rest – включає в себе контролери, які приймають дані з UI, оброблюють їх та передають на сервіс.

Пакет mail – включає в себе класи, які будують повідомлення, яке буде відправлене користувачу на вказану пошту.

Пакет until – включає в себе клас, який є генератором посилань.

Пакет dto – включає в себе класи DTO (Data Transfer Object), отриманих від користувача на контролери та передані далі на обробку до сервісу.

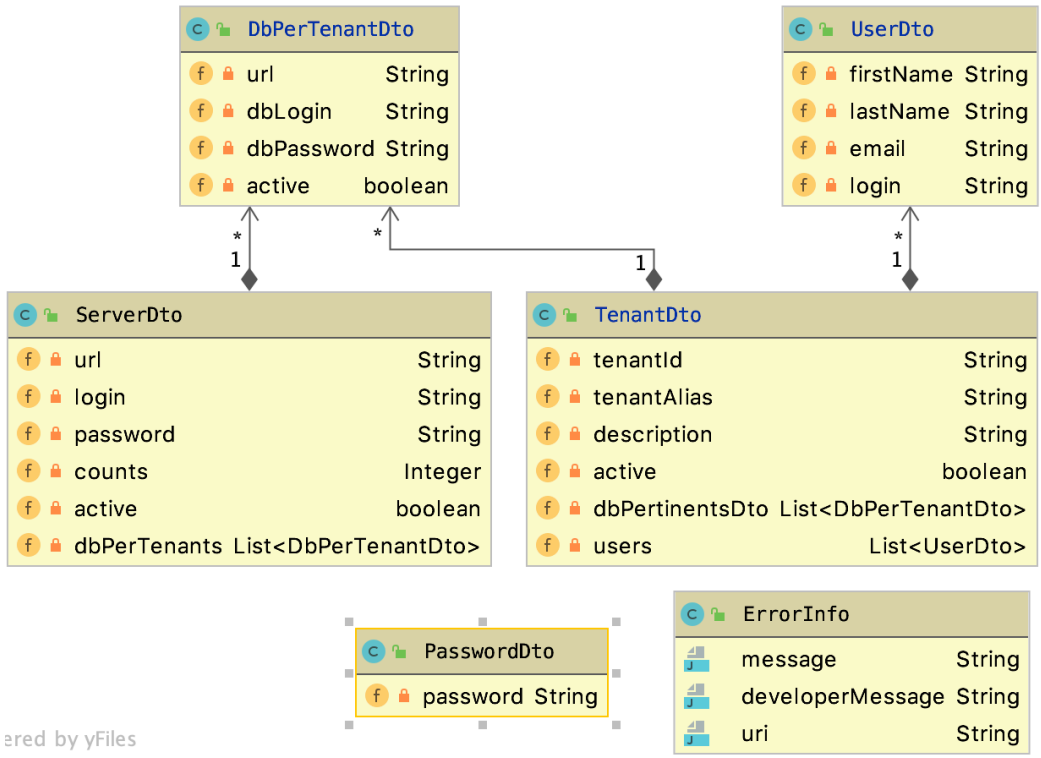

Рисунок 4.4. – Діаграма класів пакету dto додатку

Діаграма класів представлена нижче відповідає за графічне представлення класів Entity сервісу додатку, що знаходяться у пакеті «entity».

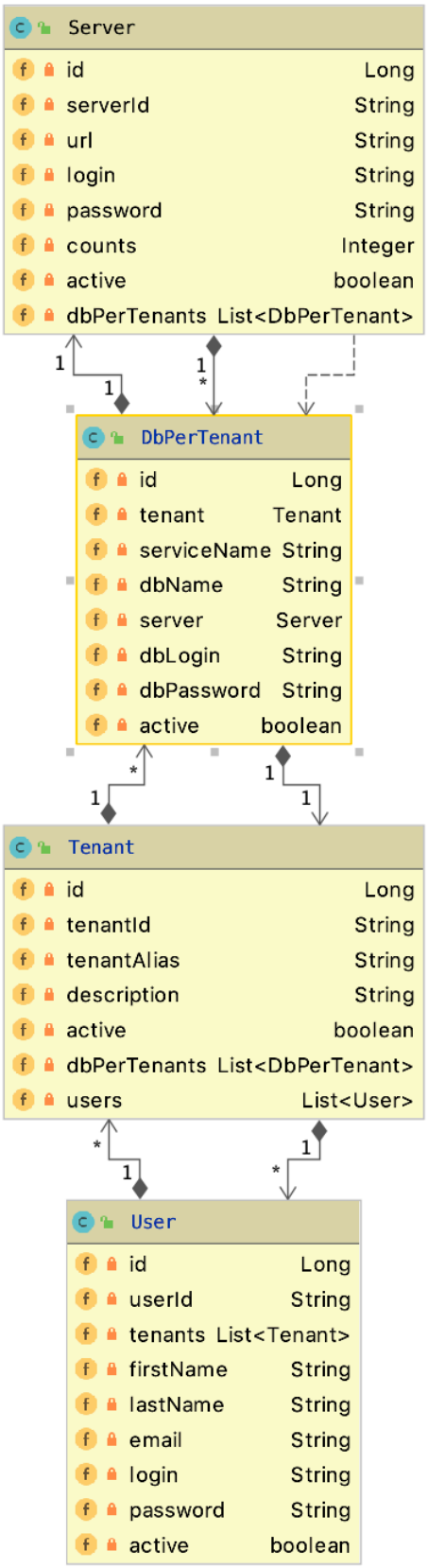

Рисунок 4.5. – Діаграма класів пакету entity сервісу додатку

Діаграма класів представлена нижче відповідає за графічне представлення контролерів сервісу додатку, що знаходяться у пакеті «rest».

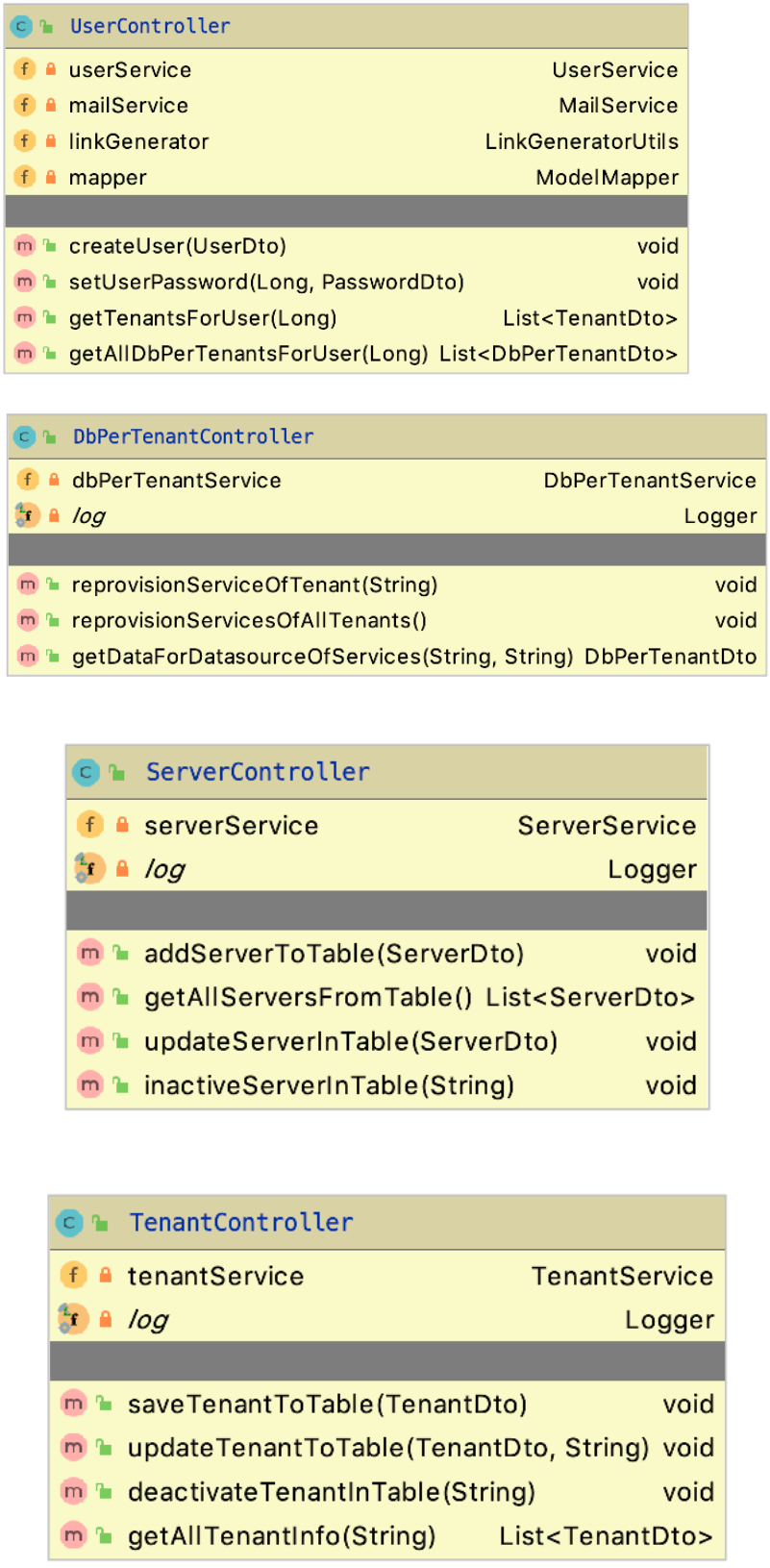

Рисунок 4.6. – 4.7 – Діаграма класів пакету rest сервісу додатку

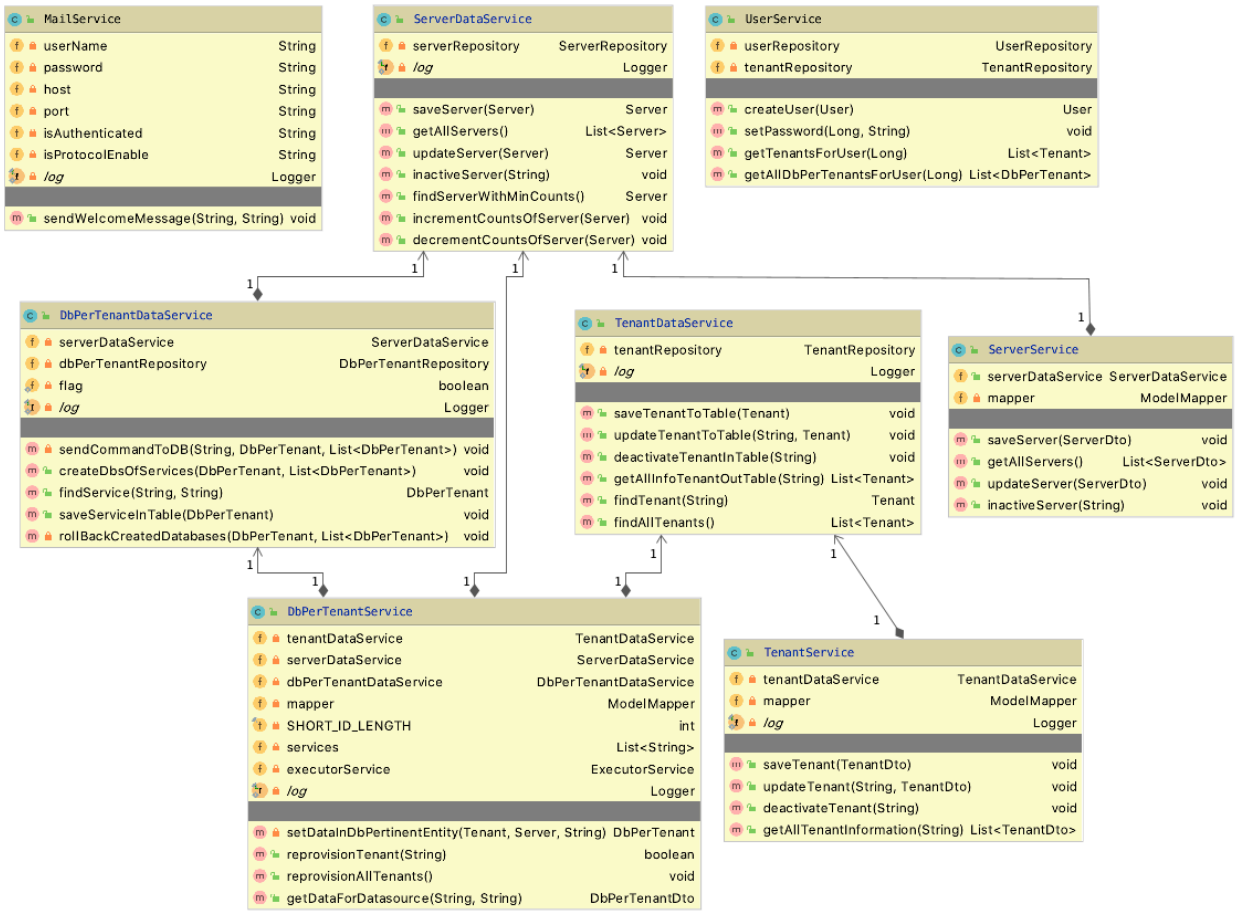

Рисунок 4.8. – Діаграма класів пакету service сервісу додатку

4.5. ОПИС REST API

REST – (Representational State Transfer) - стиль взаємодії компонентів додатка в мережі за моделлю клієнт-сервер.

Особливості архітектурного стилю:

1. Кожен запит повинен знати свій унікальний ідентифікатор - URI.
2. Запити повинні бути пов'язані між собою.
3. Для читання і зміни даних використані стандартні методи
4. Взаємодія має здійснюватися без стану

Стандартні методи REST:

GET – отримання даних без їх зміни. Це найбільш популярний і легкий метод. Він тільки повертає дані, а не змінює їх, тому на клієнті вам не потрібно піклуватися про те, що ви можете пошкодити дані.

POST – метод, який сохраненяет нової інформації.

PUT – метод, которийй змінює існуючу інформацію.

PATCH – метод, який змінює ідентифікатор існуючої інформації.

DELETE – метод, який видаляє існуючу інформацію.

При виконанні запиту до REST-сервера або формування відповіді REST-сервера, необхідно пройти певні етапи:

1. Сформувати URL
2. Поставити HTTP-заголовки
3. Вибрати тип HTTP-запиту
4. Сформувати тіло HTTP-запиту (перетворити Java об'єкти в JSON)
5. Виконати запит, скориставшись HTTP-клієнтом (наприклад, браузером)
6. Розпарсити результати запиту, тобто перетворити отриманий JSON в Java об'єкти

Давайте розглянемо опис REST API нашого додатку. Кожен запит починається з константного шляху “/api/”.

HTTP POST-запити:

http://host:port/api/user – створення нового користувача. Вхідним тілом запиту є User DTO, який має такий вигляд за допомогою JSON:

```
{
    "firstName":"Aleksandra",
    "lastName":"Ponochovna",
    "email":"oleksandra@gmail.com",
    "login":"oleksandra.ponochovna"
}
```

http://host:port/api/user/password – зміна пароля користувача, це посилання надходить користувачу на пошту. Вхідним тілом запиту є Password DTO, який має такий вигляд за допомогою JSON:

```
{
    "password":"12345sss54321"
}
```

http://host:port/api/tenant – запит на створення нового підрозділу. Вхідним тілом запиту є Tenant DTO, який має такий вигляд за допомогою JSON:

```
{
    "tenantAlias":"tenant-alias",
    "description": "description",
    "active": true,
    [ {
    "url":"some-url",
    "dbLogin":"some-db-login",
    "dbPassword":"some-db-password",
    "active":true
    }],

    [ {
    "firstName":"some- user-first-name ",
    "lastName":"some-user-last-name",
    "login":"some-user-login",
    "email":"email@gmail.com”,
    }]
}
```

http://host:port/api/{version}/db – запит на створення нового серверу, запит приймає параметр версії з application.properties, не повертає ніяких даних. Вхідним тілом запиту є Server DTO, який має такий вигляд за допомогою JSON:

```
{
        "url":"url",
        "login":"login",
        "password": "password",
        "active": true,
        "counts": 3,
        [ {
        "url":"some-url",
        "dbLogin":"some-db-login",
        "dbPassword":"some-db-password",
        "active":true
        }]
}
```

HTTP PATCH-запити:

http://host:port/api/tenant/{tenantId} – запит на обновлення вже існуючого підрозділу, запит приймає параметр зі шляху, а саме ідентифікатор підрозділу університету. Вхідним тілом запиту є Tenant DTO, який має такий вигляд за допомогою JSON:

```
{
        "tenantId":"tenant-id",
        "tenantAlias":"tenant-alias",
        "description": "description",
        "active": true,
        [ {
        "url":"some-url",
        "dbLogin":"some-db-login",
        "dbPassword":"some-db-password",
        "active":true
        }],
        [ {
```

```
        "firstName":"some- user-first-name ",
        "lastName":"some-user-last-name",
        "login":"some-user-login",
        "email":"email@gmail.com”,
        }]
}
```

HTTP PUT-запити:

http://host:port/api/{version}/db – запит на обновлення вже існуючого серверу, запит приймає параметр версії з application.properties, не повертає ніяких даних. Вхідним тілом запиту є Server DTO, який має такий вигляд за допомогою JSON:

```
{
        "url":"url",
        "login":"login",
        "password": "password",
        "active": true,
        "counts": 3,
        [ {
        "url":"some-url",
        "dbLogin":"some-db-login",
        "dbPassword":"some-db-password",
        "active":true
        }]
}
```

HTTP DELETE-запити:

http://host:port/api/tenant/{tenantId} – запит на видалення вже існуючого підрозділу, запит приймає параметр зі шляху, а саме ідентифікатор підрозділу університету.

http://host:port/api/{version}/db/{url} – запит на видалення існуючого серверу, запит приймає параметр версії з application.properties і параметр посилання, не повертає ніяких даних.

HTTP GET-запити:

http://host:port/api/reprovision/all – запит, який не приймає ні параметрів, ні тіла запиту та не має тіла відповіді, а лише HTTP код відповіді. Цей запит у результаті має те, що всі підрозділи пройшли повторний огляд (переперевірку).

http://host:port/api/reprovision/{tenantId} – запит, який приймає параметр зі шляху, а саме ідентифікатор підрозділу університету, але не приймає тіло запиту та не має тіла відповіді, а лише HTTP код відповіді. Цей запит у результаті має те, що поточний підрозділ по даному ідентифікатору пройшов повторний огляд (переперевірку).

http://host:port/api/datasource/{tenantId}/{serviceName} – запит, який приймає 2 параметри зі шляху, а саме ідентифікатор підрозділу університету та назву сервісу, але не приймає тіло запиту. Запит має тіло відповіді та HTTP код відповіді. Цей запит у результаті віддає користувачу таблицю бази даних,
що має поточний підрозділ по даному імені сервісу.

http://host:port/api/tenant/{tenantId} – запит на взяття інформації про вже існуючий підрозділ, запит приймає параметр зі шляху, а саме ідентифікатор підрозділу університету і повертає колекцію даних.

http://host:port/api/{version}/db – запит на взяття інформації про вже існуючий сервер, запит приймає параметр версії з application.properties, повертає колекцію даних.

http://host:port/api/user/{userId}/tenants – запит на взяття інформації про вже відповідні підрозділи поточного користувача, запит приймає зі шляху ідентифікатор користувача, повертає колекцію даних.

http://host:port/api/user/{userId}/tenants/{tenantId}/db-per-tenants – запит на взяття інформації про існуючі бази даних відповідного підрозділу поточного користувача, запит приймає зі шляху 2 параметри – першим параметром є ідентифікатор користувача, а другим параметром є ідентифікатор підрозділу. Повертає колекцію даних.

4.6. ВАЛІДАЦІЯ ВХІДНИХ МОДЕЛЕЙ

Таблиця 4.5. – UserDto

Назва поля	Вимоги	Опис
firstName	255 <= length >= 1	імя користувача
lastName	255 <= length >= 1	призвіще користувача
email	255 <= length >= 1	пошта користувача
userId	255 <= length >= 1	зовнішний ідентифікатор користувача
login	255 <= length >= 1	логін користувача

Таблиця 4.6. – PasswordDto

Назва поля	Вимоги	Опис
password	255 <= length >= 1	пароль користувача

Таблиця 4.7. – ServerDto

Назва поля	Вимоги	Опис
url	255 <= length >= 1	посилання на поточних серверу
login	255 <= length >= 1	логін доступу до поточного серверу
password	255 <= length >= 1	пароль доступу до поточного серверу

counts	має бути заданий	кількість створених баз даних
dbPertinentsDto	size >= 1	колекція баз даних серверу поточного підрозділу
active		по дефолту є false

Таблиця 4.8. – TenantDto

Назва поля	Вимоги	Опис
tenantId	255 <= length >= 1	зовнішний ідентифікатор підрозділу
tenantAlias	255 <= length >= 1	назва підрозділу
description	255 <= length >= 1	опис підрозділу
dbPertinentsDto	size >= 1	колекція баз даних серверу поточного підрозділу
users	size >= 1	колекція користувачів поточного підрозділу
active		по дефолту є false

Таблиця 4.8. – DbPerTenantDto

Назва поля	Вимоги	Опис
url	255 <= length >= 1	посилання на базу даних серверу
dbPassword	255 <= length >= 1	пароль до бази даних
dbLogin	255 <= length >= 1	логін до бази даних
active		по дефолту є false

5. РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

Для початку роботи з нашим додатком, необхідно відкрити браузер (будь-який, що підтримує HTML5) та перейти за посиланням tms.service/ru.

При повній загрузці сайту, ми бачимо початкову сторінку, де є дві запропоновані кнопки для старту роботи нашої системи.

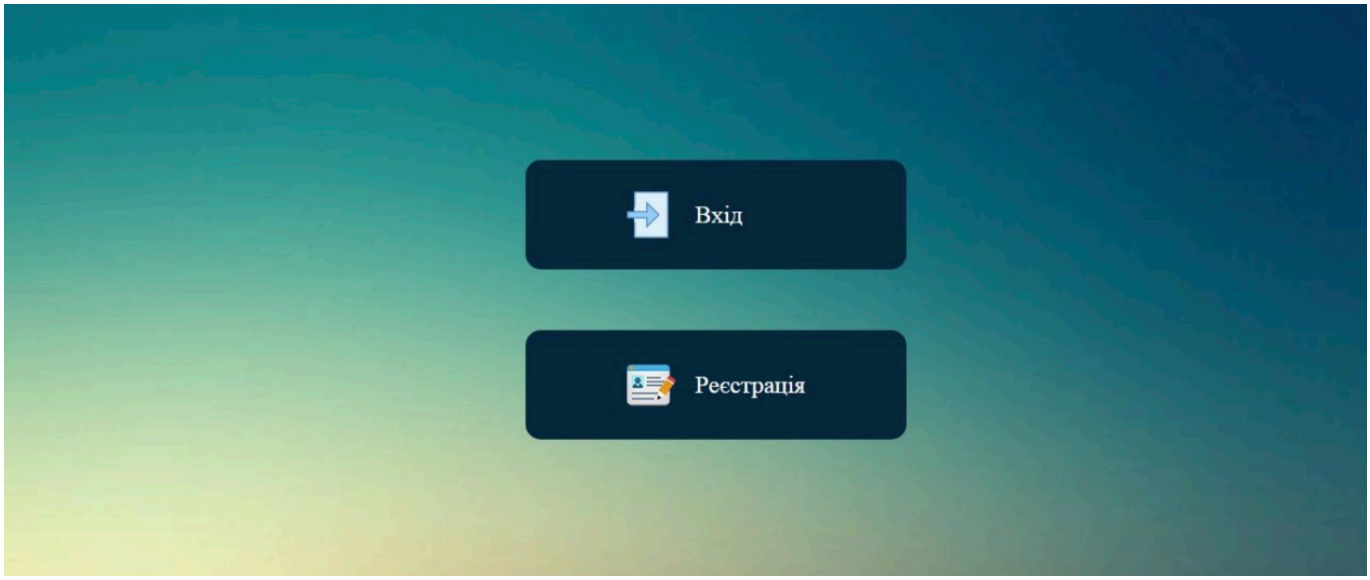

Рисунок 5.1. – Початкова сторінка додатку

Для продовження роботи – необхідно клікнути на активне вікно «Вхід» - якщо ви вже були зареєстровані в системі, або «Реєстрація» - задля створення нового користувача в системі додатку.

При відкритті активного вікна «Вхід», що представлений на рисунку 5.2., ми бачимо форму, яка необхідна для заповнення. Поле «email» та «пароль» не мають бути пустими, а користувач с такими даними повинен бути зареєстрований у системі. Поле «email» має містити коректні дані пошти, інакше система буде видавати помилки.

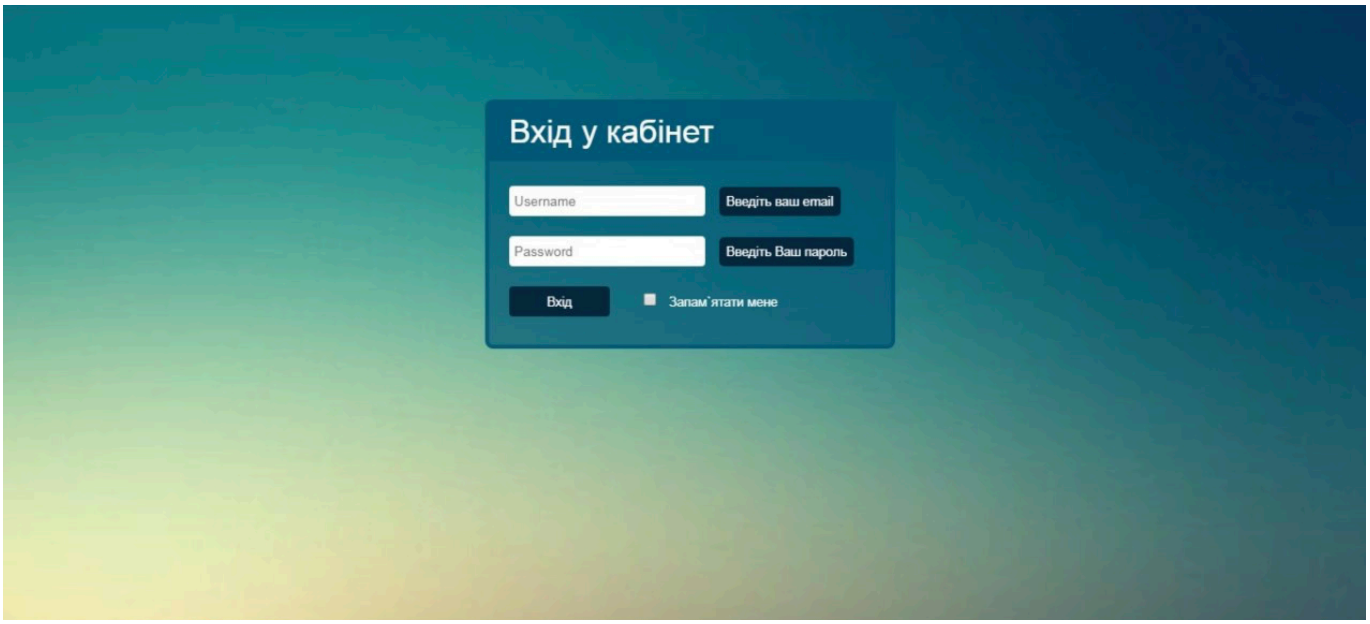

Рисунок 5.2. – Сторінка входу у системі

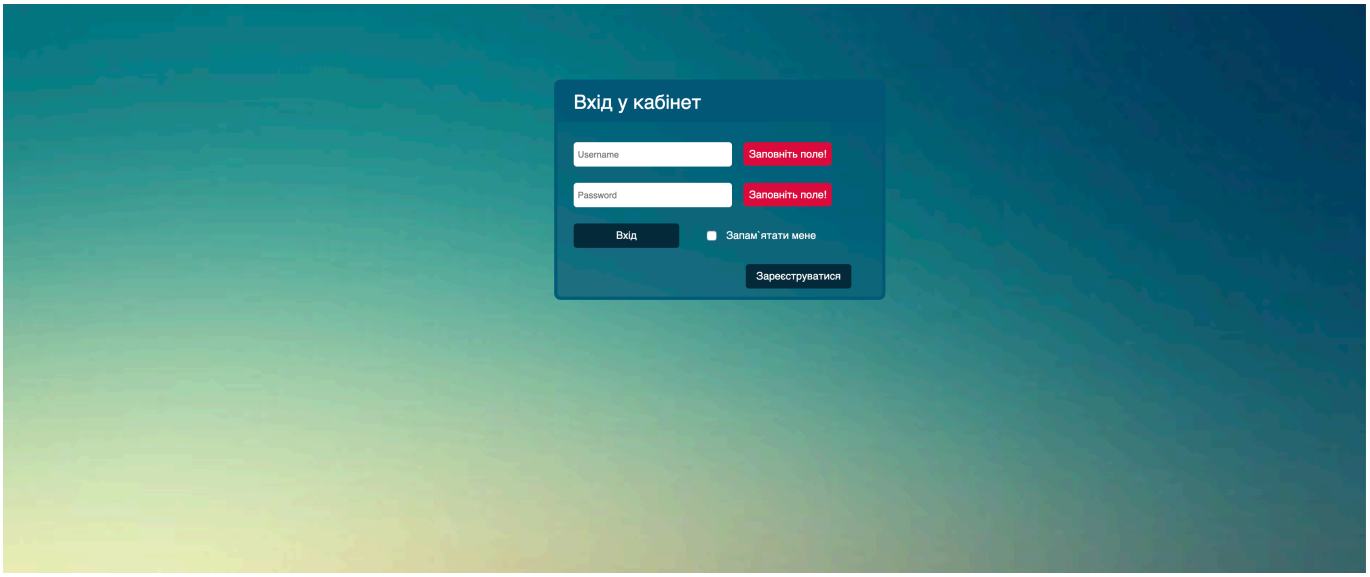

Рисунок 5.3. – Відображення помилок при пустих полях

Після успішного логіну – користувачу висвічується заголовок, що авторизація пройшла успішно. Це показано на рисунку 5.4. нижче. Після цього користувач буде переадресований на певну сторінку, це залежить від його ролі у додатку.

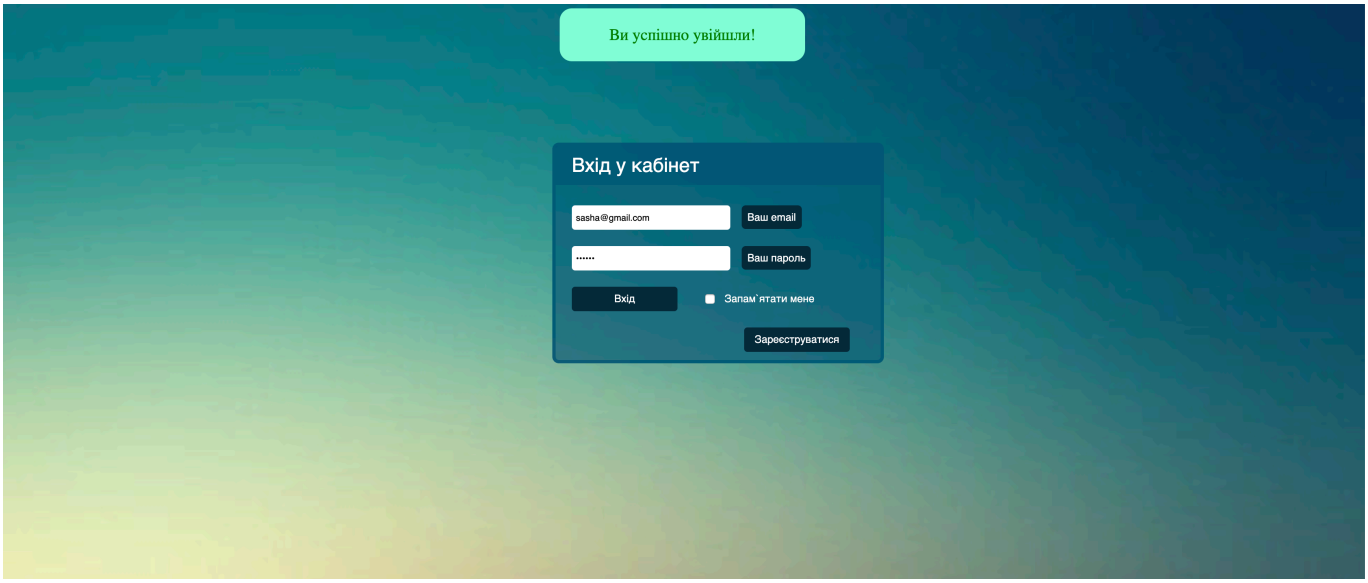

Рисунок 5.4. – Повідомлення про успішний логін користувача

При відкритті вікна «Реєстрація», що є активним на рисунку 5.1. (початкова сторінка додатку) – ми бачимо форму, яка також необхідна для заповнення. Вікно «Реєстрація» представлене на рисунку 5.5.. Поле «ім'я», «прізвище» та «email» не мають бути пустими, а користувач с такими даними не має бути зареєстрований у системі. Поле «email» також має містити коректні дані пошти, інакше система буде видавати помилки, що показані на рисунках 5.6. и 5.7.

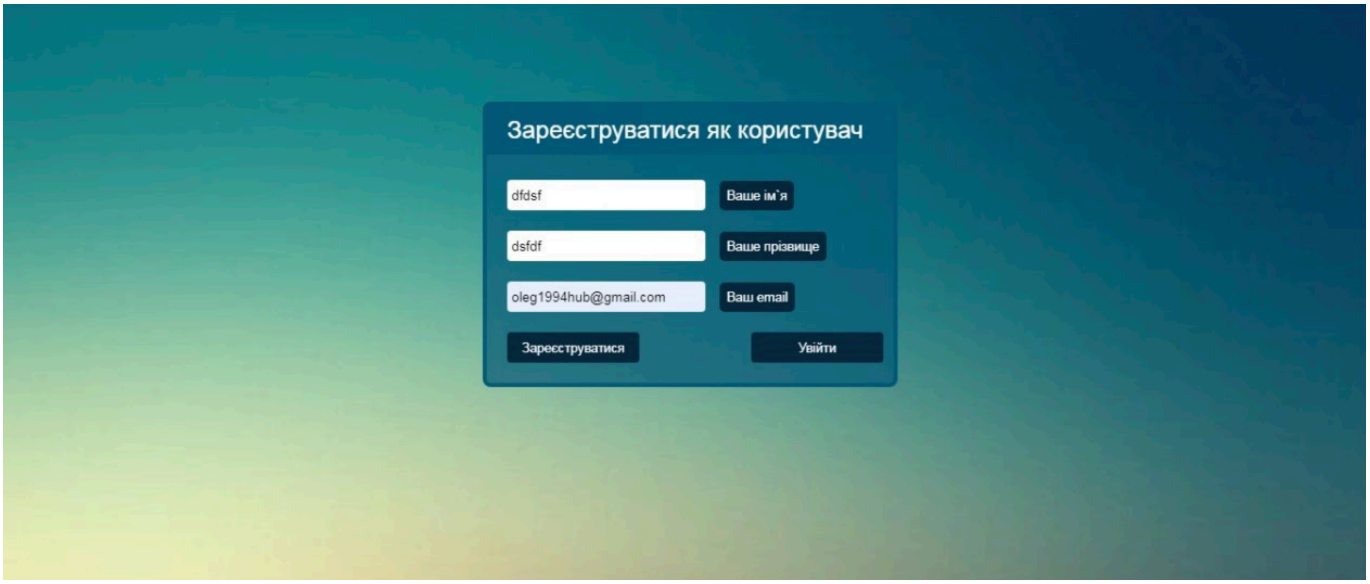

Рисунок 5.5. – Сторінка реєстрації у систему

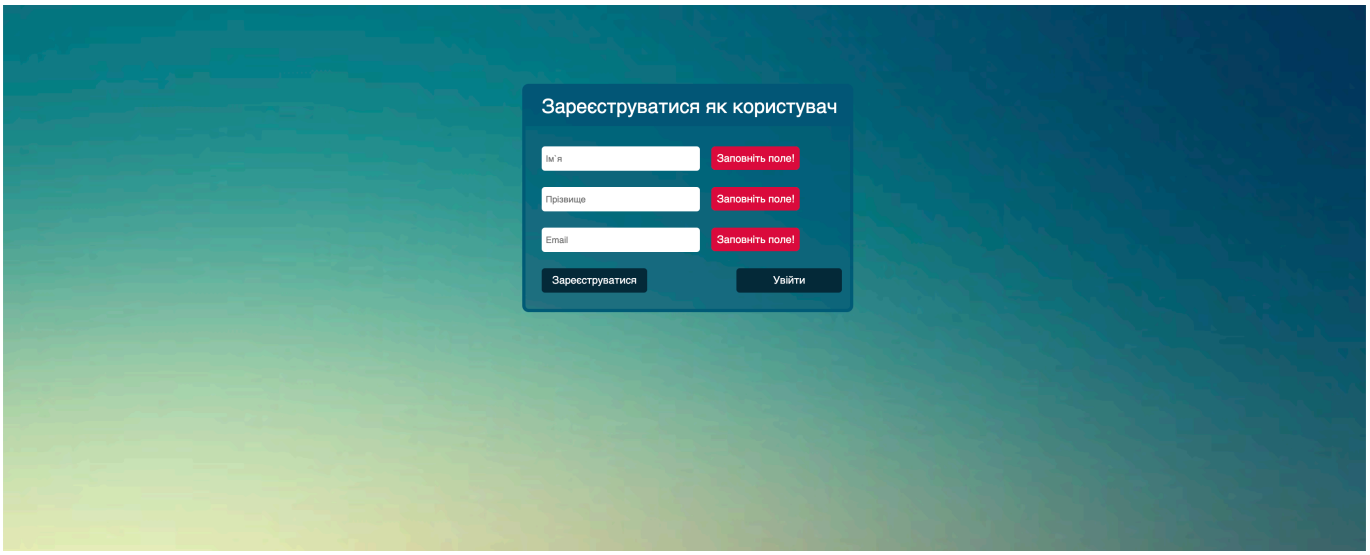

Рисунок 5.6. – Помилки, які є при введенні пустих значень у формі

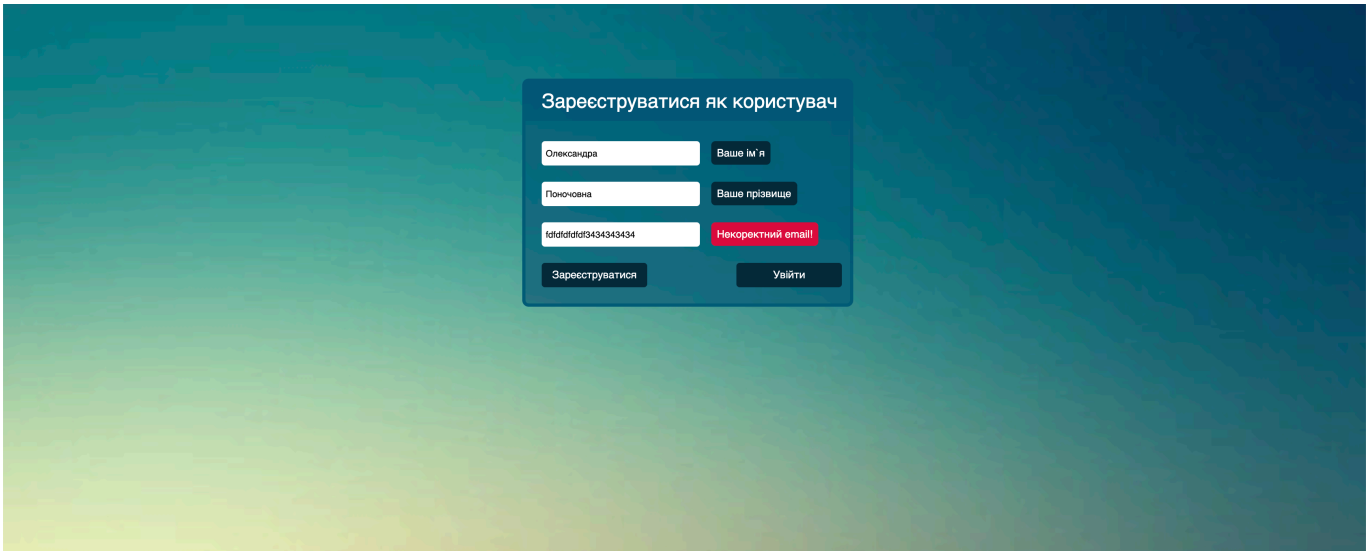

Рисунок 5.7. – Помилка при некоректному введенні пошти у формі

У створеній системі є 2 типи користувачів:

1. Адміністратор

Адміністратор має такі права, як: переглядати список усіх існуючих серверів, активувати/деактивувати деякі з них, переглядати список усіх зареєстрованих підрозділів та їх активні бази даних, активувати/деактивувати деякі з них. Також адміністратор має право надіслати запрошення по пошті користувачам для запрошення у систему. Також адміністратор має право створення нового підрозділу.

2. Звичайний користувач

Звичайний користувач має такі права, як: переглядати список усіх існуючих баз даних відповідних підрозділів, а також активність даних баз. Також користувач має право створення нового підрозділу, після чого – даний користувач буде адміністратором цього підрозділу.

На далі, після успішного в ходу у систему, розпочнемо з ролі Адміністратора та представимо на рисунках нижче вікна, які будуть доступні користувачу з цією роллю.

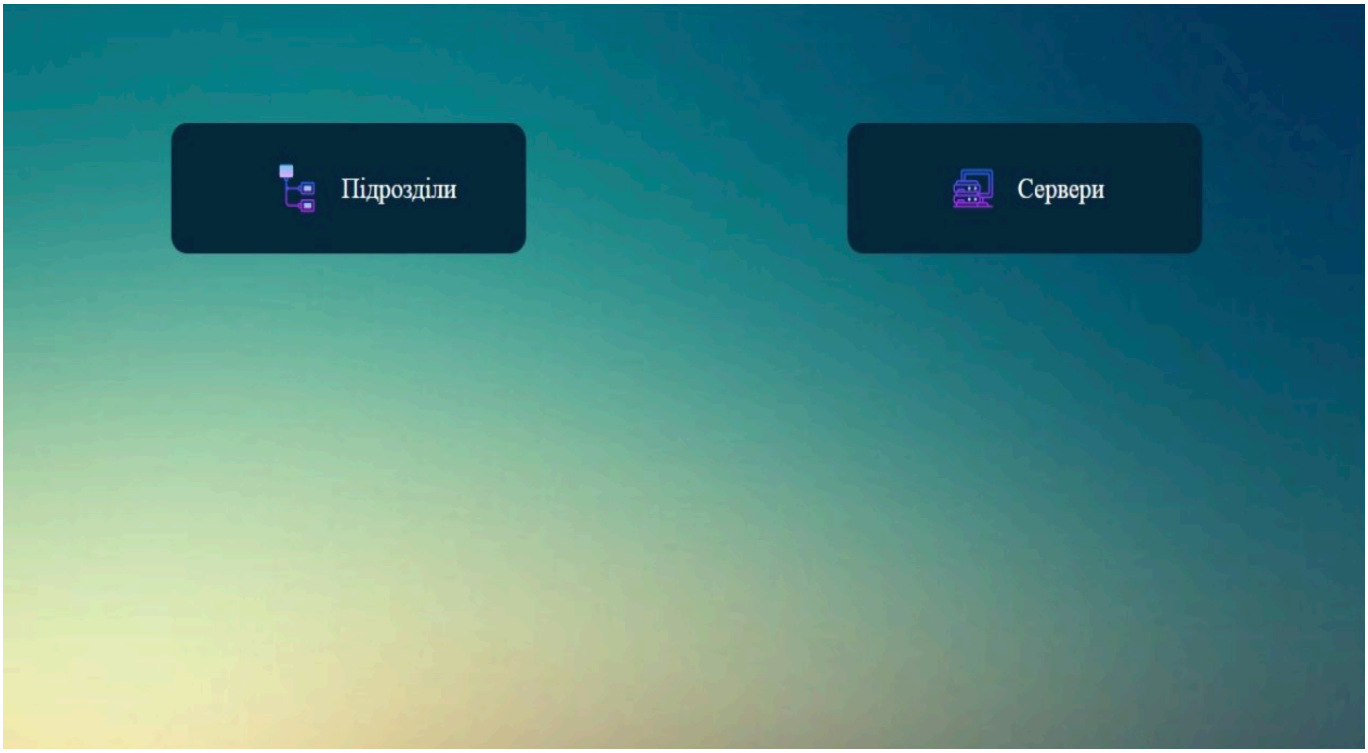

Рисунок 5.8. – Вікно після логіну для користувача з роллю Адміністратор

Тут є 2 активних вікна для кліку: «Підрозділи» та «Сервери». Розпочнемо зі сторінки «Підрозділи». На цій сторінці буде зображено список усіх підрозділів, а також доступ до баз даних кожного з них.

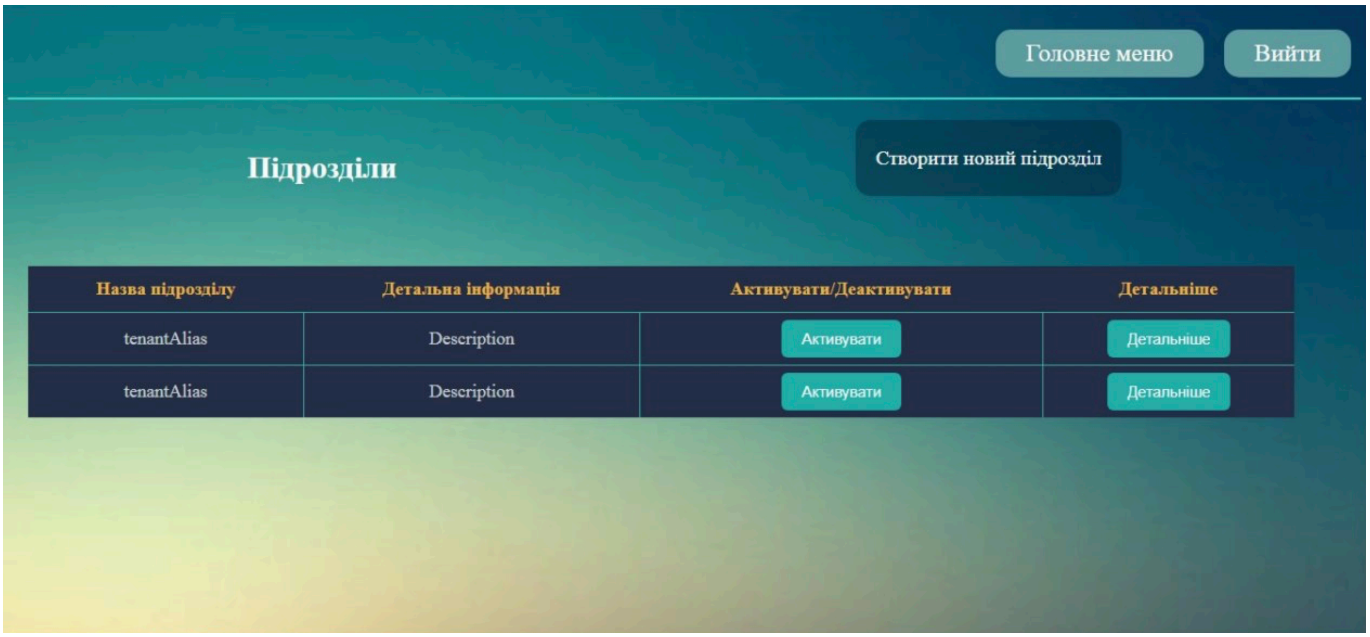

Рисунок 5.9. – Список активних підрозділів для користувача з роллю Адміністратор

Тут ми бачимо дані про існуючі підрозділу. З активних дій можемо виділити кнопку «Створити новий підрозділ», якщо клікнути на дану кнопку, то ми побачимо форму як на рисунку 5.10. нижче. У формі необхідно ввести назву підрозділу та детальну інформацію.

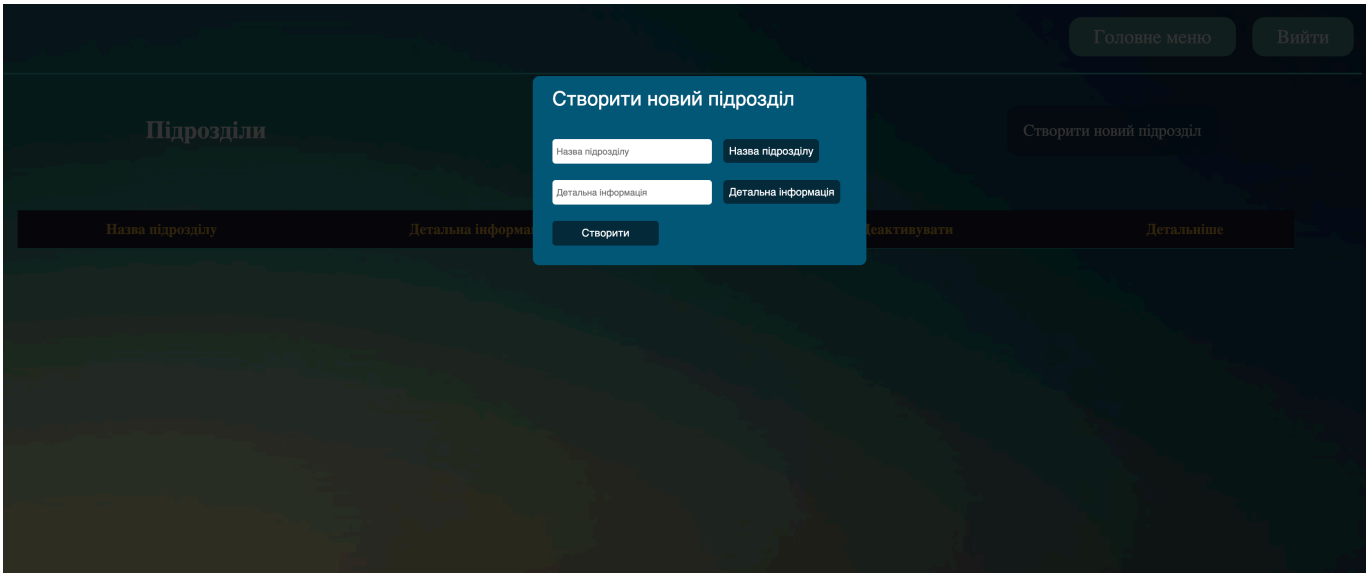

Рисунок 5.10. – Вікно для створення нового підрозділів для користувача з роллю Адміністратор

Повернемося до таблиці «Підрозділи», що зазначена на рисунку 5.9. У таблиці є активні кнопки «Активувати/Деактивувати», задля активація або деактивації певного підрозділу. А також кнопка «Детальніше» біля кожного з існуючих підрозділів.

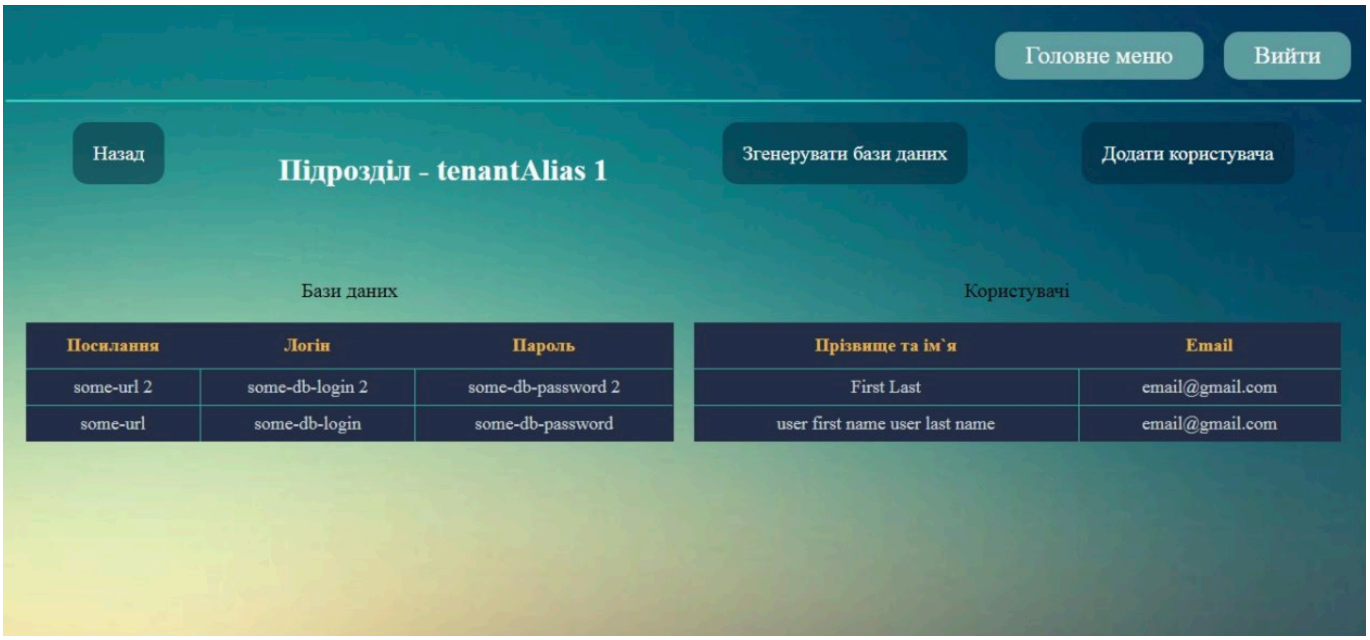

Рисунок 5.11. – Вікно детальної інформації підрозділу для користувача з роллю Адміністратор

На цій активній сторінці – ми маємо дані про даний підрозділ, а саме: активні бази даних та користувачі даного підрозділу.

У правому верхньому кутку сторінки є кнопка «Додати користувача».

Адміністратор, натиснувши цю кнопку, переходить до форми, де він має вписати пошту користувача, який має бути створеним. На рисунку 5.13. зображена дана форма.

Після того, як Адміністратор введе пошту та натисне «Додати» – на дану пошту буде надісланий лист, що зображений на рисунку 5.12.

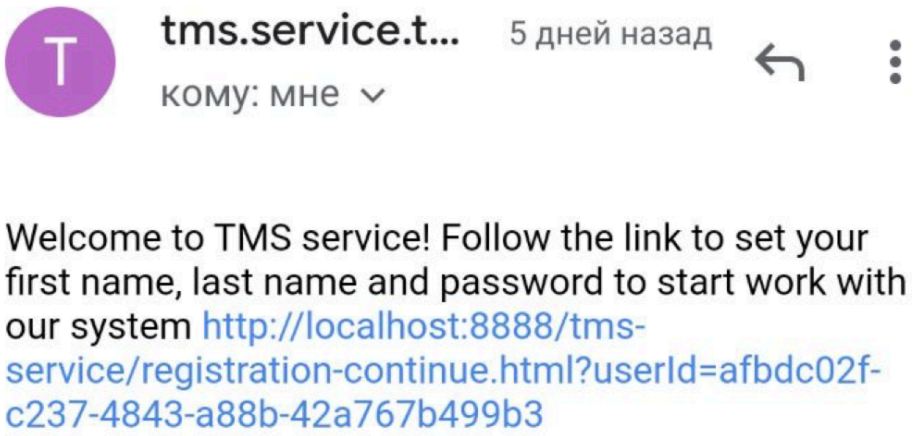

Рисунок 5.12. – Лист з посиланням на реєстрацію

На рисунку 5.12. лист з посиланням на форму реєстрації (зображена на рисунку 5.14.), щоб користувач ввів свої дані, а саме своє ім’я, призвіще та придумав пароль для користування системою.

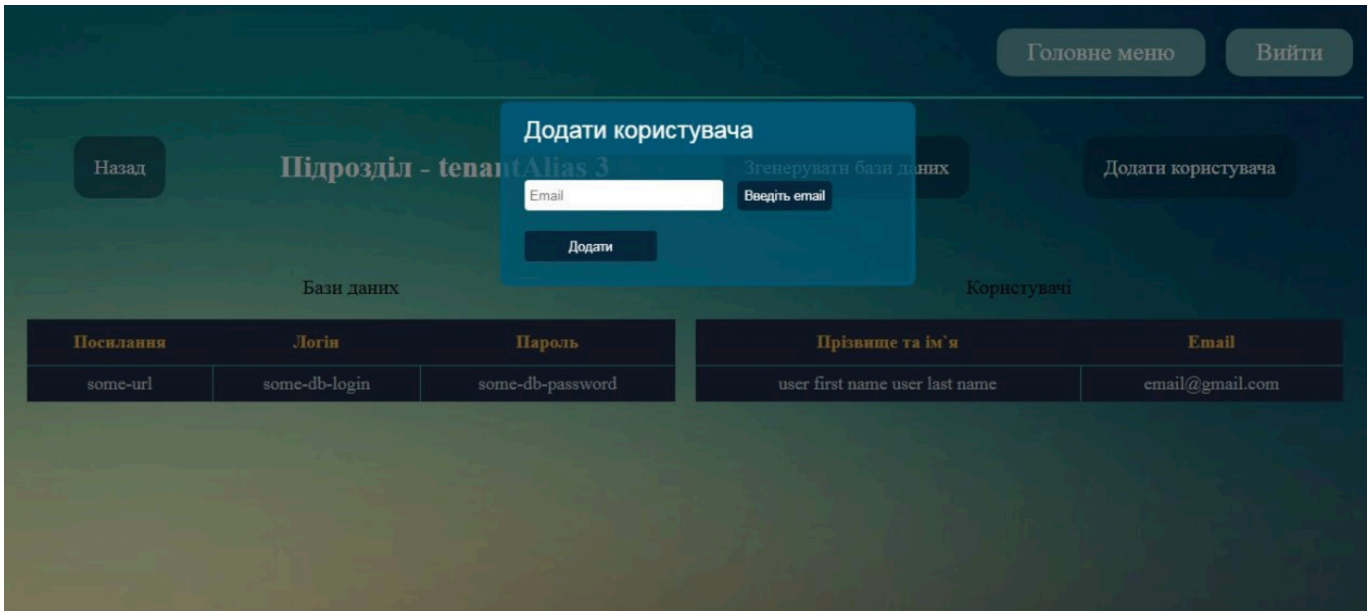

Рисунок 5.13. – Вікно для додавання користувача у системі для користувача з роллю Адміністратор

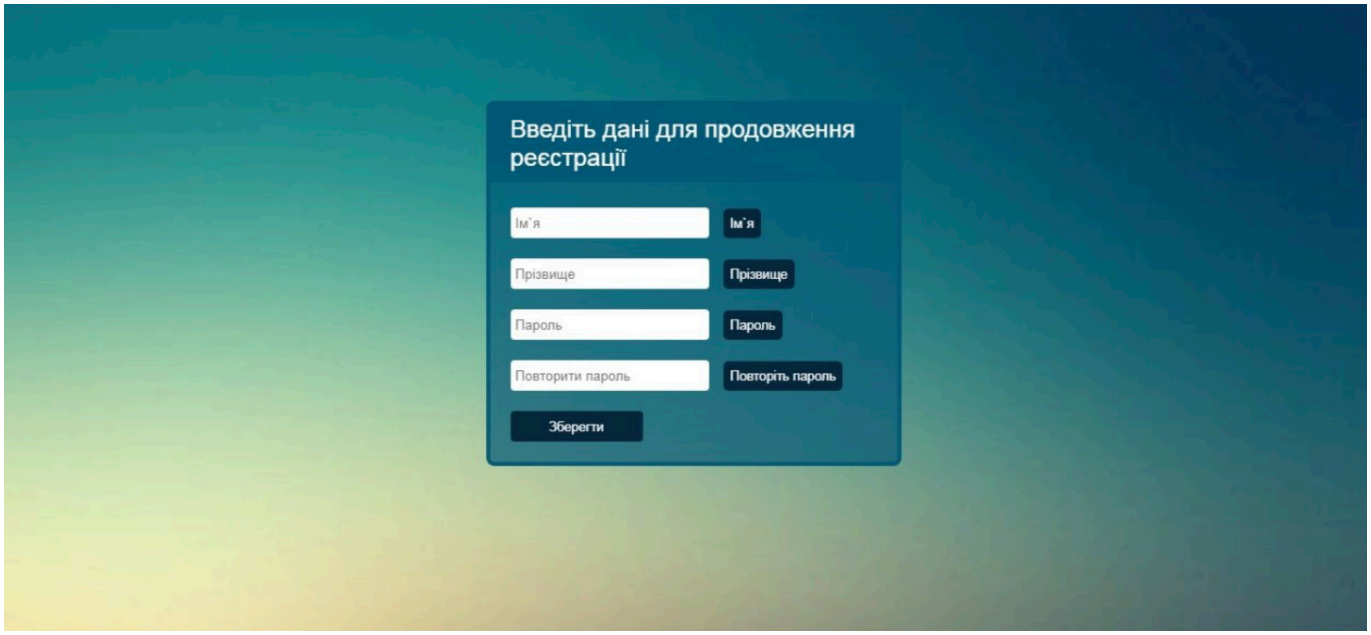

Рисунок 5.14. – Форма введення даних для продовження реєстрації як звичайний користувач

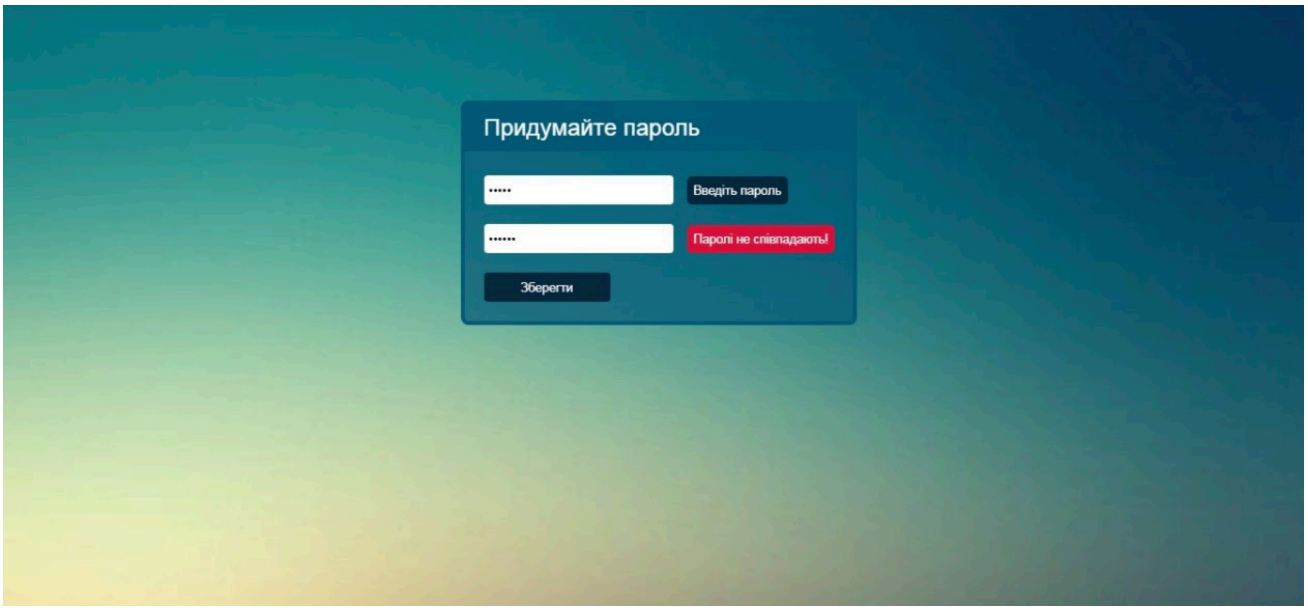

Рисунок 5.15. – Форма введення даних для продовження реєстрації як звичайний користувач

Повернемося до рисунку 5.8. та розглянемо сторінку «Сервери». Тут ми бачимо дані про існуючі сервери (рисунок 5.16.). З активних дій можемо виділити кнопку «Створити новий сервер», якщо клікнути на дану кнопку, то ми побачимо форму як на рисунку 5.17. нижче. У формі необхідно ввести посилання, логін та пароль.

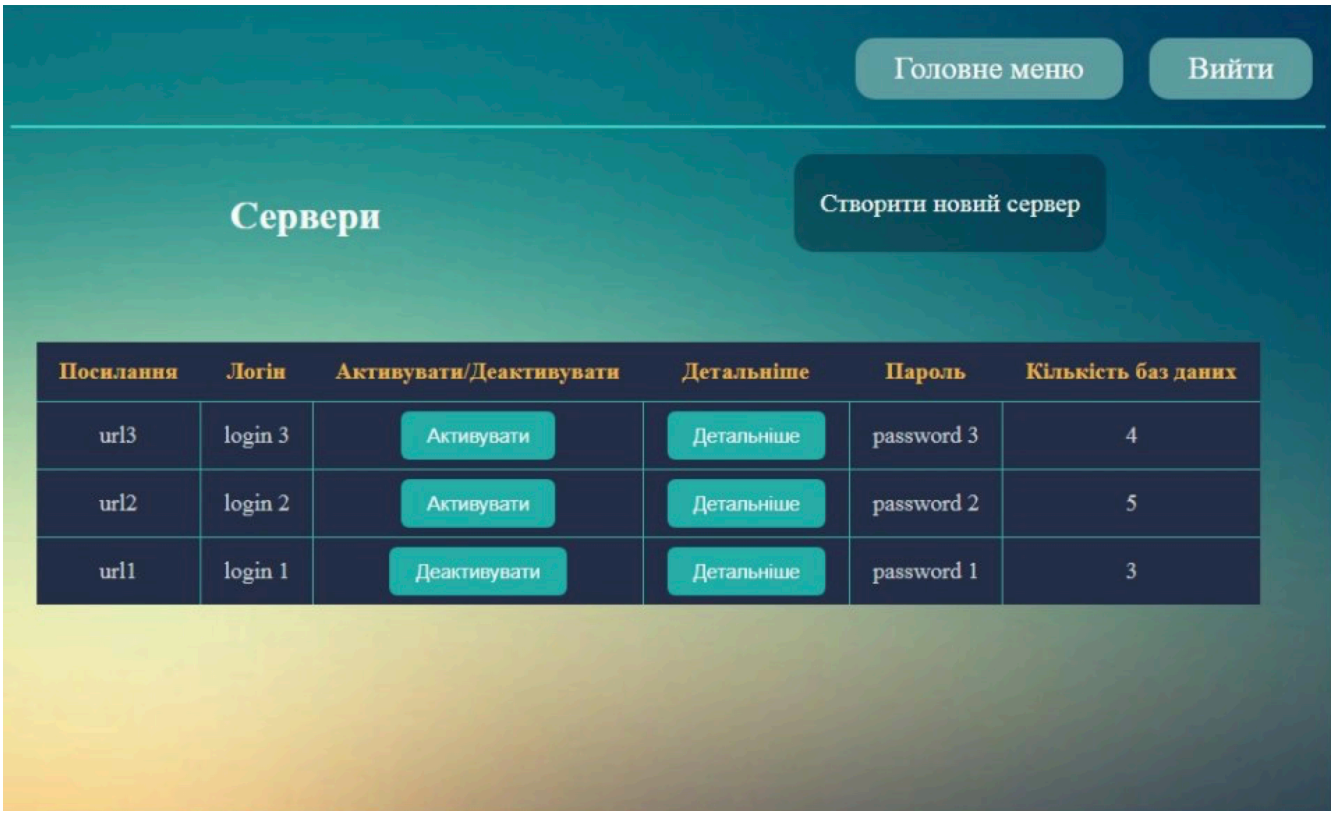

Рисунок 5.16. – Список активних серверів для користувача з роллю Адміністратор

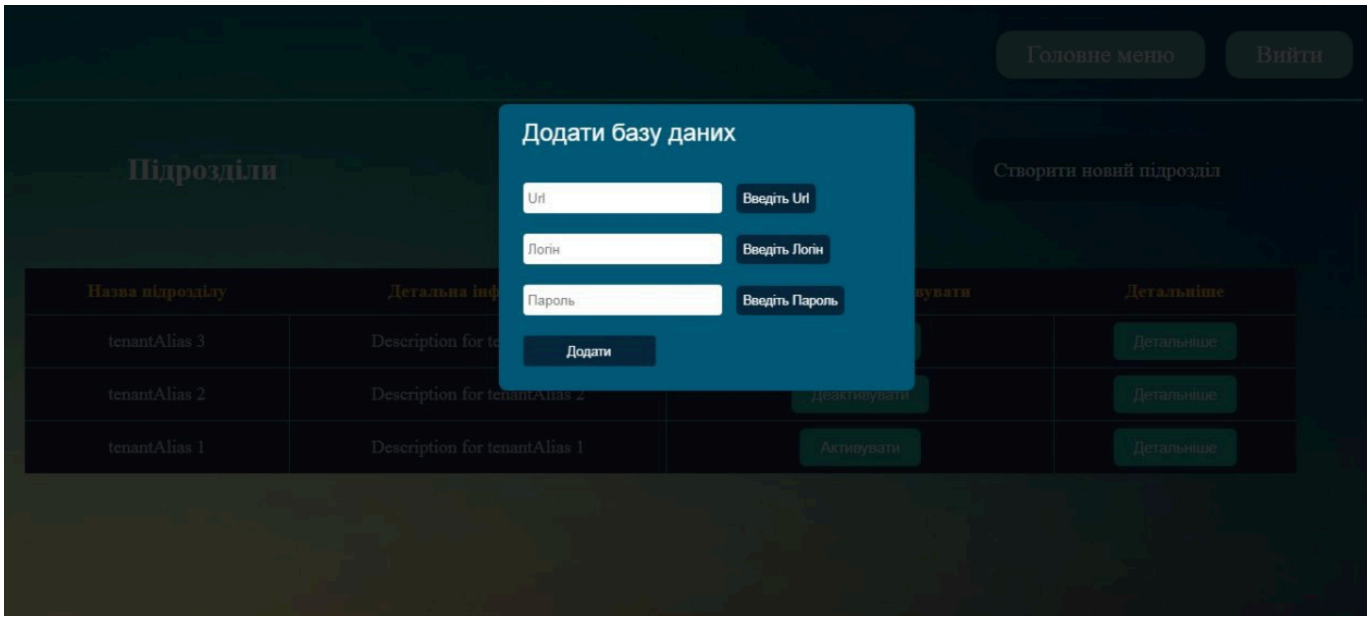

Рисунок 5.17. – Створення нового серверу для користувача з роллю Адміністратор

Повернемося до таблиці «Сервери», що зазначена на рисунку 5.16. У таблиці є активні кнопки «Активувати/Деактивувати», задля активація або деактивації певного серверу. А також кнопка «Детальніше» біля кожного з існуючих серверів.

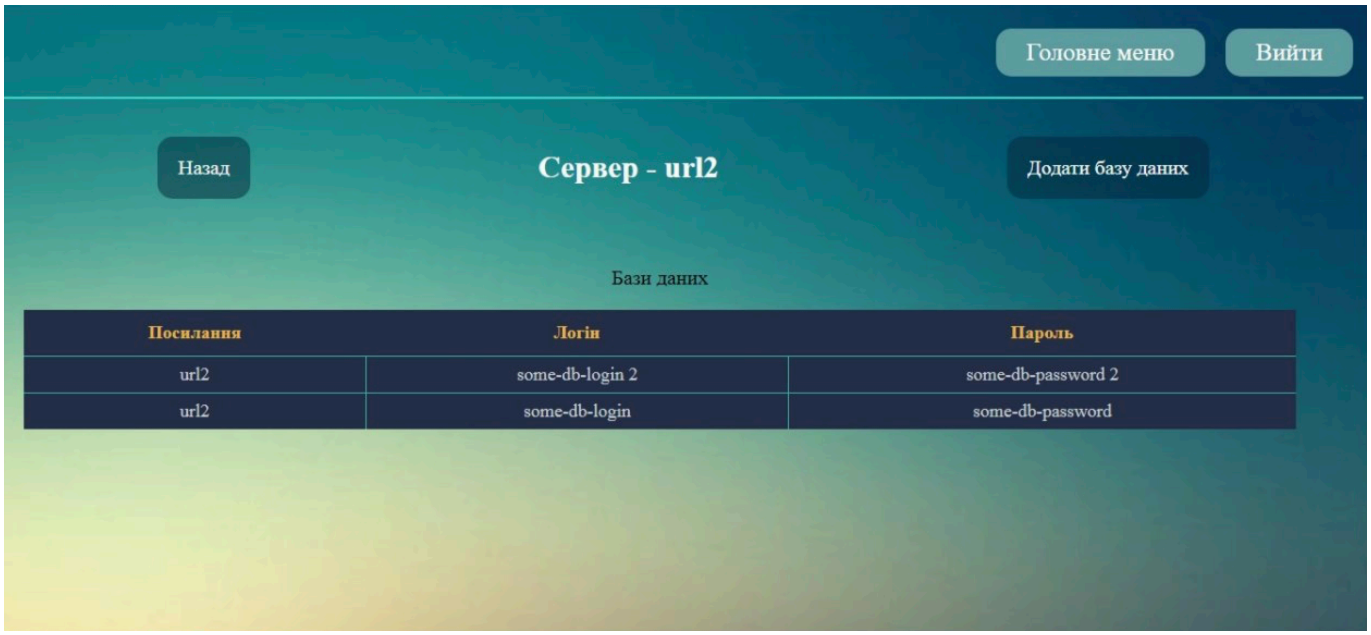

Рисунок 5.18. – Детальна інформація вибраного серверу для користувача з роллю Адміністратор

На цій активній сторінці (рис. 5.18) – ми маємо дані про даний підрозділ, а саме: активні бази даних даного серверу.

Тепер розглянемо як виглядає інтерфейс системи для звичайного користувача системи.

Після успішного логіну – ми бачимо сторінку, що зображена на рисунку 5.19., а саме – поточні бази даних підрозділів, що зареєстрували користувачів. У правому верхньому кутку є кнопка «Створити новий підрозділ», натиснувши яку користувач переходить до форми створення нового підрозділу, що зображена на рисунку 5.20.

Для створення нового підрозділу – користувачу необхідно ввести назву підрозділу та детальну інформація. Після того, як Адміністратор зазначить даний підрозділ активним – можна буде генерувати бази даних.

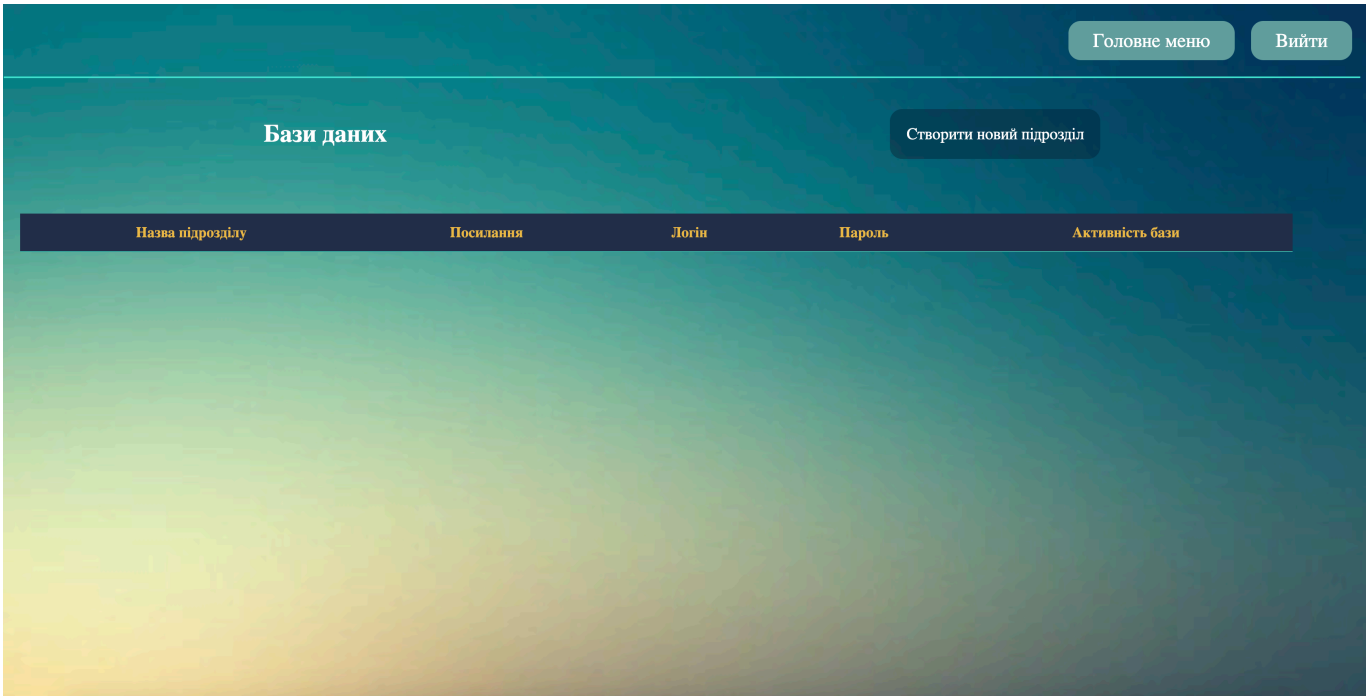

Рисунок 5.19. – Детальна інформація баз даних підрозділів для звичайного користувача

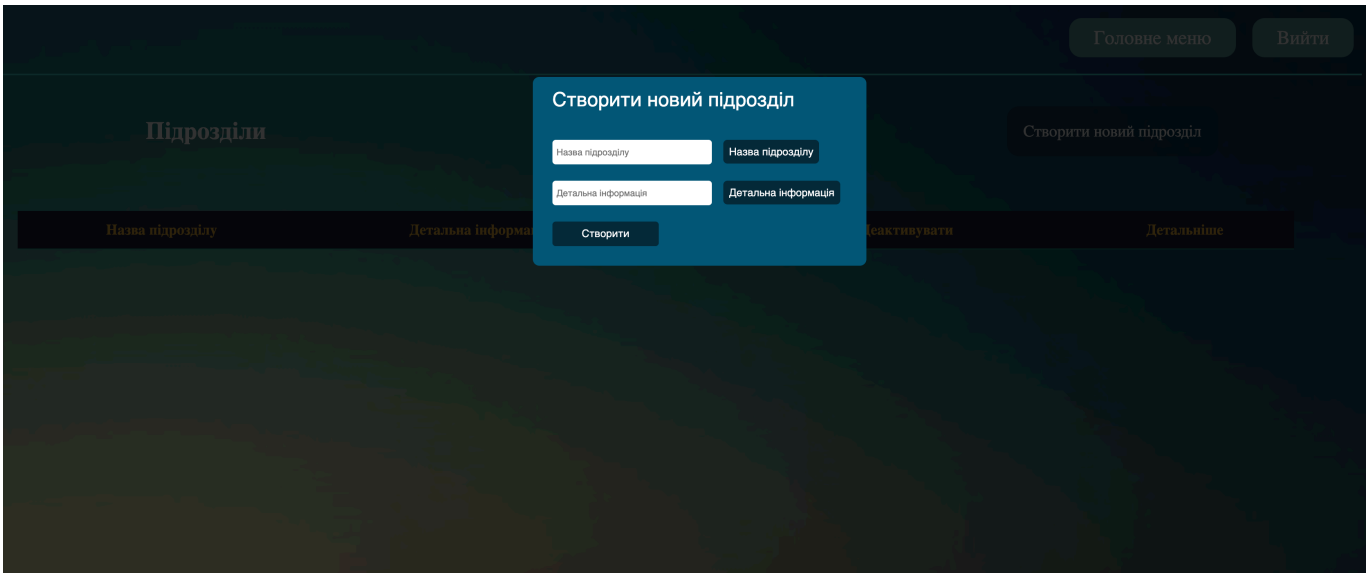

Рисунок 5.20. – Форма створення нового підрозділу для звичайного користувача

Отже, ось таким є процес роботи з створеною системою. Процес не є трудомістким, але зберігає не прості колекції даних. Навігація системою є легкою та доступною для будь–якого користувача, незалежно від його навичок користування ПК.

ВИСНОВКИ

У ході виконання даної роботи було розроблено систему, що дозволяє зберігати підрозділи університету. В даній дипломній роботі було досліджено основні концепції побудови додатків на базі мікросервісної архітектури, а також розглянуто основні особливості, переваги та недоліки даного підходу до побудови програмних систем, порівняно з класичним монолітним рішенням. Звичайно, монолітна архітектура чудово розв’язує певні задачі, але із зростанням складності, використовування цього виду архітектури вже не зможе якісно вирішувати свої функції. Через це так стрімко розвинулася мікросервісна архітектура.

Для написання додатку було обрано методи та засоби реалізації – JavaScript, HTML, CSS, MySQL, Bootstrap, Java, Maven, Spring MVC, Spring Security, Spring Boot.

В даному проекті було розглянуто основні принципи проектування даних систем. Також було розглянуто ключові компоненти для побудови мікросервісних систем, які надають необхідну гнучкість всій працюючій системі.

Як результат всіх проведених досліджень, було розроблено тестову програму, яка підтверджує концепцію даної архітектури та демонструє її працездатність..

Підсумовуючи, можна стверджувати, що мікросервісна архітектура має право на існування, але не є кращим архітектурним рішенням для побудови програмних систем. Більше того, мікросервіси не рекомендовано створювати без глибокого попереднього аналізу предметної області та чіткого виділення обмежених контекстів. Звичайно, різноманітні методи аналізу предметної області були

представлені у даній роботі задля покращення отриманих результатів від вибору систему, а також пропонується створювати мікросервіси на базі існуючого моноліту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ньюмен С. Создание микросервисов / Ньюмен С. – СПб.: Питер, 2016 – 304с.
2. Офіційний сайт Java Micro Services. – Режим доступу: http://cppmicroservices.org/ – Дата доступу: 29.05.2020
3. Офіційний сайт Pistache framework. – Режим доступу: http://pistache.io/ Дата доступу: 29.05.2020
4. Офіційний сайт Spring framework. – Режим доступу: https://spring.io – Дата доступу: 29.05.2020
5. 3. Chris Richardson. From Design to Deployment / Chris Richardson, Floyd Smith, 2016. – 74 p.
6. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf – Addison-Wesley, 2004 – 736 p.
7. E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software Evans – Addison-Wesley, 2003 – 560 p.
8. I. Nadareishvili. Microservice Architecture: Aligning Principles, Practices, and Culture / I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen – O'Reilly Media, 2016 – 146 p.
9. Introduction to microservices. – Режим доступу: https://nginx.com/blog/introduction-to-microservices/ – Дата доступу: 29.05. 2020
10. Martin Fowler – Microservices – Режим доступу: http://martinfowler.com/articles/microservices.html – Дата доступу: 28.05.2017
11. Using an API Gateway. – Режим доступу https://nginx.com/blog/buildingmicroservices-using-an-api-gateway/ – Дата доступу: 27.05.2020

Додаток А

Мікросервіс створення профілю підрозділу університету

Специфікація УКР. НТУУ “КПІ імені Ігоря Сікорського”

Аркушів 1

2020

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ ”КПІ імені Ігоря Сікорського”_ТЕФ_АПЕПС_ТР62_20Б	Записка	Пояснювальна записка
Компоненти		
УКР.НТУУ ”КПІ імені Ігоря Сікорського”_ТЕФ_АПЕПС_ТР62_20Б 1	Текст програмного модулю	

Додаток Б

Мікросервіс створення профілю підрозділу університету

Текст програмного модулю

Специфікація УКР. НТУУ “КПІ імені Ігоря Сікорського”

Аркушів 10

2020

```
@Slf4j
@RestController
@CrossOrigin(origins = "${cors.allowed.origins}", maxAge = 1800)
public class AuthController {

  @Autowired
  private AuthenticationManager authenticationManager;
  @Autowired
  private UserService userService;
  @Autowired
  private PasswordEncoder passwordEncoder;

  @PreAuthorize("permitAll()")
  @PostMapping(LOGIN)
  public SuccessedAuthenticationDto login(@RequestBody @Valid CredentialsDto
request) {
    try {
      authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(request.getEmail(), request.getPassword()));
      User user = userService.findUserByEmail(request.getEmail());
      return user != null ? SuccessedAuthenticationDto.builder()
          .role(user.getRole().name())
          .userId(user.getUserId())
          .build() : null;
    } catch (AuthenticationException e) {
      throw new BadCredentialsException("Invalid username/password");
    }
  }

  @PreAuthorize("permitAll()")
  @PostMapping(value = TmsEndpointsMapping.USER_PASSWORD, consumes =
"application/json", produces = "application/json")
  @ResponseStatus(HttpStatus.OK)
  public void setPasswordByLinkFromEmail(@PathVariable("userId") String userId,
@RequestBody @Valid PasswordDto request) {
    userService.setPassword(userId, passwordEncoder.encode(request.getPassword()));
  }

  @PreAuthorize("permitAll()")
  @PostMapping(value = TmsEndpointsMapping.USER_DATA, consumes = "application/json",
produces = "application/json")
  @ResponseStatus(HttpStatus.OK)
  public void setUserDataByLinkFromEmail(@PathVariable("userId") String userId,
@RequestBody @Valid UserDataDto request) {
    String encodedPassword = passwordEncoder.encode(request.getPassword());
    request.setPassword(encodedPassword);
    userService.setUserData(userId, request);
  }
}
package com.agroculture.tms.rest;

import com.agroculture.tms.dto.DbPerTenantDto;
import com.agroculture.tms.dto.TmsEndpointsMapping;
import com.agroculture.tms.service.controllerService.DbPerTenantService;
import java.util.List;
import javax.validation.constraints.NotBlank;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
```

```
@Slf4j
@CrossOrigin(origins = "${cors.allowed.origins}", maxAge = 1800)
public class DbPerTenantController {

  @Autowired
  private DbPerTenantService dbPerTenantService;

  @GetMapping(value = TmsEndpointsMapping.REPROVISION_TENANT, consumes =
"application/json", produces = "application/json")
  public void reprovisionServiceOfTenant(@NotBlank @PathVariable("tenantId") String
tenantId) {
    dbPerTenantService.reprovisionTenant(tenantId);
    log.info("Tenant with TenantId [{}] has been reprovision.", tenantId);
  }

  @GetMapping(value = TmsEndpointsMapping.REPROVISION_ALL, consumes =
"application/json", produces = "application/json")
  public void reprovisionServicesOfAllTenants() {
    dbPerTenantService.reprovisionAllTenants();
    log.info("All Tenants have been reprovision");
  }

  @GetMapping(value = TmsEndpointsMapping.DATASOURCE_TENANT, consumes =
"application/json", produces = "application/json")
  public DbPerTenantDto getDataForDatasourceOfServices(@PathVariable("tenantId")
String tenantId) {
    return dbPerTenantService.getDataForDatasource(tenantId);
  }
}
package com.agroculture.tms.rest;

import com.agroculture.tms.dto.ServerDto;
import com.agroculture.tms.dto.TmsEndpointsMapping;
import com.agroculture.tms.service.controllerService.ServerService;
import java.util.List;
import javax.validation.Valid;
import javax.validation.constraints.NotBlank;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@Slf4j
@CrossOrigin(origins = "${cors.allowed.origins}", maxAge = 1800)
public class ServerController {

  @Autowired
  private ServerService serverService;

  @PostMapping(value = TmsEndpointsMapping.DATABASE, consumes = "application/json",
produces = "application/json")
  public void addServerToTable(@Valid @RequestBody ServerDto serverDto) {
    serverService.saveServer(serverDto);
    log.info("Server with url [{}] has been saved.", serverDto.getUrl());
  }

  @GetMapping(value = TmsEndpointsMapping.DATABASE, consumes = "application/json",
produces = "application/json")
  public List<ServerDto> getAllServersFromTable() {
    return serverService.getAllServers();
  }
```

```
  @PutMapping(value = TmsEndpointsMapping.DATABASE, consumes = "application/json",
produces = "application/json")
  public void updateServerInTable(@Valid @RequestBody ServerDto serverDto) {
    serverService.updateServer(serverDto);
    log.info("Server with url [{}] has been updated.", serverDto.getUrl());
  }

  @DeleteMapping(value = TmsEndpointsMapping.DATABASE_URL, consumes =
"application/json", produces = "application/json")
  public void inactiveServerInTable(@NotBlank @PathVariable("url") String url) {
    serverService.inactiveServer(url);
    log.info("Server with url [{}] has been deactivated", url);
  }

  @PutMapping(value = TmsEndpointsMapping.DATABASE_URL, consumes =
"application/json", produces = "application/json")
  public void activeServerInTable(@NotBlank @PathVariable("url") String url) {
    serverService.activeServer(url);
    log.info("Server with url [{}] has been deactivated", url);
  }

  @GetMapping(value = TmsEndpointsMapping.DATABASE_URL, consumes =
"application/json", produces = "application/json")
  public ServerDto getServerByUrl(@PathVariable String url) {
    return serverService.getServerByUrl(url);
  }
}
package com.agroculture.tms.rest;

import com.agroculture.tms.dto.TenantDto;
import com.agroculture.tms.dto.TmsEndpointsMapping;
import com.agroculture.tms.repository.DbPerTenantRepository;
import com.agroculture.tms.repository.ServerRepository;
import com.agroculture.tms.service.controllerService.TenantService;
import java.util.List;
import java.util.stream.Collectors;
import javax.validation.Valid;
import lombok.extern.slf4j.Slf4j;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@Slf4j
@CrossOrigin(origins = "${cors.allowed.origins}", maxAge = 1800)
public class TenantController {

  private TenantService tenantService;
  @Autowired
  private DbPerTenantRepository dbPerTenantRepository;
  @Autowired
  private ServerRepository serverRepository;

  @Autowired
  private ModelMapper mapper;

  public TenantController(TenantService tenantService) {
    this.tenantService = tenantService;
  }

  @PostMapping(value = TmsEndpointsMapping.TENANTS, consumes = "application/json",
produces = "application/json")
```

```
  public void saveTenantToTable(@Valid @RequestBody TenantDto tenantDto) {
    tenantService.saveTenant(tenantDto);
    log.info("Tenant with alias: [{}], has been saved", tenantDto.getTenantAlias());
  }

  @PatchMapping(value = TmsEndpointsMapping.TENANT, consumes = "application/json",
produces = "application/json")
  public void updateTenantToTable(@Valid @RequestBody TenantDto tenantDto,
@PathVariable("tenantId") String tenantId) {
    tenantService.updateTenant(tenantId, tenantDto);
    log.info("Tenant: [{}], has been update", tenantId);
  }

  @DeleteMapping(value = TmsEndpointsMapping.TENANT, consumes = "application/json",
produces = "application/json")
  public void deactivateTenantInTable(@PathVariable("tenantId") String tenantId) {
    tenantService.deactivateTenant(tenantId);
    log.info("Tenant: [{}], has been deactivating", tenantId);
  }

  @PutMapping(value = TmsEndpointsMapping.TENANT, consumes = "application/json",
produces = "application/json")
  public void activateTenantInTable(@PathVariable("tenantId") String tenantId) {
    tenantService.activateTenant(tenantId);
    log.info("Tenant: [{}], has been activating", tenantId);
  }

  @GetMapping(value = TmsEndpointsMapping.TENANT, consumes = "application/json",
produces = "application/json")
  @ResponseBody
  public TenantDto getAllTenantInfo(@PathVariable("tenantId") String tenantId) {
    TenantDto tenant = tenantService.getTenantInformation(tenantId);
    tenant.getDbPertinentsDto().forEach(db -> {
      db.setUrl(db.getDbName());
    });
    return tenant;
  }

  @GetMapping(value = TmsEndpointsMapping.TENANTS_BY_USER, consumes =
"application/json", produces = "application/json")
  public List<TenantDto> getTenantsForUser(@PathVariable("userId") String userId) {
    return tenantService.getTenantsForUser(userId).stream()
        .map(tenant -> mapper.map(tenant, TenantDto.class))
        .collect(Collectors.toList());
  }

  @GetMapping(value = TmsEndpointsMapping.TENANTS, consumes = "application/json",
produces = "application/json")
  @ResponseBody
  public List<TenantDto> getAllTenants() {
    return tenantService.getAllTenants();
  }
}

package com.agroculture.tms.rest;

import com.agroculture.tms.dto.*;
import com.agroculture.tms.entity.User;
import com.agroculture.tms.service.controllerService.UserService;
import java.util.List;
import java.util.stream.Collectors;
import javax.validation.Valid;
```

```
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@CrossOrigin(origins = "${cors.allowed.origins}", maxAge = 1800)
public class UserController {

  @Autowired
  private UserService userService;
  @Autowired
  private ModelMapper mapper;

  @PostMapping(value = TmsEndpointsMapping.USER, consumes = "application/json",
produces = "application/json")
  public void createUser(@Valid @RequestBody UserDto userDto) {
    userService.createUser(mapper.map(userDto, User.class));
  }

  @PostMapping(value = TmsEndpointsMapping.CREATE_USER_BY_EMAIL, consumes =
"application/json", produces = "application/json")
  public void createUserByEmail(@PathVariable String email,
@PathVariable("tenantId") String tenantId) {
    userService.createUserByEmail(email, tenantId);
  }

  @GetMapping(value = TmsEndpointsMapping.USER_TENANTS, consumes =
"application/json", produces = "application/json")
  public List<TenantDto> getTenantsForUser(@PathVariable("userId") String userId) {
    return userService.getTenantsForUser(userId).stream()
        .map(tenant -> mapper.map(tenant, TenantDto.class))
        .collect(Collectors.toList());
  }

  @GetMapping(value = TmsEndpointsMapping.USER_DB_PER_TENANTS, consumes =
"application/json", produces = "application/json")
  public List<DbPerTenantDto> getDbPerTenantsForUser(@PathVariable("userId") String
userId) {
    return userService.getAllDbPerTenantsForUser(userId).stream()
        .map(tenant -> mapper.map(tenant, DbPerTenantDto.class))
        .collect(Collectors.toList());
  }
}
package com.agroculture.tms.service.controllerService;

import com.agroculture.tms.dto.DbPerTenantDto;
import com.agroculture.tms.entity.*;
import com.agroculture.tms.exceptions.DbPerTenantNotExist;
import com.agroculture.tms.service.dataService.*;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;
import lombok.extern.slf4j.Slf4j;
import org.apache.commons.lang3.RandomStringUtils;
import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

@Service
@Slf4j
public class DbPerTenantService {
```

```
  @Autowired
  private TenantDataService tenantDataService;
  @Autowired
  private ServerDataService serverDataService;
  @Autowired
  private DbPerTenantDataService dbPerTenantDataService;
  @Autowired
  private ModelMapper mapper;

  final private int SHORT_ID_LENGTH = 6;

  @Value("#{'${application.services}'.split(',')}")
  private List<String> services = new ArrayList<>();

  private ExecutorService executorService = Executors.newFixedThreadPool(100);

  private DbPerTenant setDataInDbPertinentEntity(Tenant tenant, Server server) {
    DbPerTenant newDbPerTenant = new DbPerTenant();
    newDbPerTenant.setTenant(tenant);
    newDbPerTenant.setDbName(tenant.getTenantAlias() + "_service_" +
RandomStringUtils.randomAlphanumeric(SHORT_ID_LENGTH));
    newDbPerTenant.setServer(server);
    newDbPerTenant.setDbLogin(tenant.getTenantAlias() + "_log_" +
RandomStringUtils.randomAlphanumeric(SHORT_ID_LENGTH));
    newDbPerTenant.setDbPassword(tenant.getTenantAlias() + "_pass_" +
RandomStringUtils.randomAlphanumeric(SHORT_ID_LENGTH));
    newDbPerTenant.setActive(true);
    return newDbPerTenant;
  }

  public boolean reprovisionTenant(String tenantId) {
    List<DbPerTenant> dbPerTenantListForDelete = new ArrayList<>();
    Tenant tenant = tenantDataService.findTenant(tenantId);
    Server server = serverDataService.findServerWithMinCounts();

    DbPerTenant newDbPerTenant = dbPerTenantDataService.findService(tenantId);
    if (newDbPerTenant == null) {
      newDbPerTenant = setDataInDbPertinentEntity(tenant, server);
      dbPerTenantListForDelete.add(newDbPerTenant);
      dbPerTenantDataService.createDbsOfServices(newDbPerTenant,
dbPerTenantListForDelete);
      dbPerTenantDataService.saveServiceInTable(newDbPerTenant);
      serverDataService.incrementCountsOfServer(server);
    }
    return true;
  }

  public void reprovisionAllTenants() {
    List<Tenant> tenantList = tenantDataService.findAllTenants();
    List<Future> futureList = new ArrayList<>();
    try {
      for (Tenant list : tenantList) {
        Runnable task = () -> reprovisionTenant(list.getTenantId());
        futureList.add(executorService.submit(task));
      }
      for (Future list : futureList) {
        list.get();
      }
    } catch (InterruptedException e) {
      log.error("Interrupted exception");
    } catch (ExecutionException e) {
```

```
      log.error("Execution exception");
    } finally {
      executorService.shutdown();
    }
  }

  public DbPerTenantDto getDataForDatasource(String tenantId) {
    DbPerTenant newDbPerTenant = dbPerTenantDataService.findService(tenantId);
    if (newDbPerTenant != null) {
      String url = newDbPerTenant.getServer().getUrl() + "/" +
newDbPerTenant.getDbName();
      DbPerTenantDto dbPerTenantDto = mapper.map(newDbPerTenant,
DbPerTenantDto.class);
      dbPerTenantDto.setUrl(url);
      return dbPerTenantDto;
    } else {
      String thisMethodName = new Object() {
      }.getClass().getEnclosingMethod().getName();
      String mess = String.format("%s: DbPerTenant with tenantId %s and serviceName
%s is not exist", thisMethodName, tenantId);
      log.info(mess);
      throw new DbPerTenantNotExist(mess);
    }
  }
}


@Service
@Slf4j
public class DbPerTenantDataService {

  @Autowired
  private ServerDataService serverDataService;

  private DbPerTenantRepository dbPerTenantRepository;

  public DbPerTenantDataService(DbPerTenantRepository dbPerTenantRepository) {
    this.dbPerTenantRepository = dbPerTenantRepository;
  }

  private static boolean flag;

  private void sendCommandToDB(String command, DbPerTenant newDbPerTenant,
List<DbPerTenant> dbPerTenantListForDelete) {
    Connection con = null;
    Statement stm = null;
    String thisMethodName = new Object() {
    }.getClass().getEnclosingMethod().getName();
    try {
      con = DriverManager.getConnection(newDbPerTenant.getServer().getUrl(),
          newDbPerTenant.getServer().getLogin(),
          newDbPerTenant.getServer().getPassword());
      stm = con.createStatement();
      stm.executeUpdate(command);
      log.info("Connection is successful");
    } catch (Exception err) {
      if (flag) {
        log.error("[{}]: Server or user were not created! Was called ROLLBACK",
thisMethodName);
        rollBackCreatedDatabases(newDbPerTenant, dbPerTenantListForDelete);
      } else {
        String mess = String.format("%s: It was problem with ROLLBACK for Databases
```

```
of Services", thisMethodName);
        log.error(mess);
        throw new DatabaseWasNotCreated(mess);
      }
    } finally {
      try {
        if (stm != null) {
          stm.close();
          con.close();
        }
      } catch (SQLException err) {
        String mess = String.format("%s: Connection can not be closed",
thisMethodName);
        log.error(mess);
        throw new ConnectionCanNotBeClosed(mess);
      }
    }
  }

  @Transactional
  public void createDbsOfServices(DbPerTenant newDbPerTenant, List<DbPerTenant>
dbPerTenantListForDelete) {
    String command = String.format("CREATE DATABASE %s; CREATE USER %s WITH PASSWORD
'%s'",
        newDbPerTenant.getDbName(),
        newDbPerTenant.getDbLogin(), newDbPerTenant.getDbPassword());
    flag = true;
    sendCommandToDB(command, newDbPerTenant, dbPerTenantListForDelete);
    String thisMethodName = new Object() {
    }.getClass().getEnclosingMethod().getName();
    log.info("[{}]: Server [{}] and user [{}] were successful created",
thisMethodName, newDbPerTenant.getDbName(), newDbPerTenant.getDbLogin());
  }

  public DbPerTenant findService(String tenantId) {
    List<DbPerTenant> listOfDbPerTenant =
dbPerTenantRepository.findByTenant_TenantId(tenantId);
    return listOfDbPerTenant.isEmpty() ? null : listOfDbPerTenant.get(0);
  }

  @Transactional
  public void saveServiceInTable(DbPerTenant newDbPerTenant) {
    dbPerTenantRepository.save(newDbPerTenant);
    String thisMethodName = new Object() {
    }.getClass().getEnclosingMethod().getName();
    log.info("[{}]: Tenant [{}] were successful saved in the table", thisMethodName,
        newDbPerTenant.getTenant().getTenantId());
  }

  private void rollBackCreatedDatabases(DbPerTenant newDbPerTenant,
List<DbPerTenant> dbPerTenantListForDelete) {
    for (DbPerTenant dbPerTenantForDelete : dbPerTenantListForDelete) {
      String command = String.format("DROP DATABASE %s",
dbPerTenantForDelete.getDbName());
      flag = false;
      sendCommandToDB(command, newDbPerTenant, dbPerTenantListForDelete);
      String thisMethodName = new Object() {
      }.getClass().getEnclosingMethod().getName();
      log.info("[{}]: Server [{}] has been deleted", thisMethodName,
dbPerTenantForDelete.getDbName());
      serverDataService.decrementCountsOfServer(newDbPerTenant.getServer());
    }
```

```
  }
}
package com.agroculture.tms.service.dataService;

import static java.util.UUID.randomUUID;

import com.agroculture.tms.entity.Server;
import com.agroculture.tms.exceptions.ServerAlreadyExist;
import com.agroculture.tms.exceptions.ServerNotExist;
import com.agroculture.tms.repository.ServerRepository;
import java.util.List;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Slf4j
@Service
public class ServerDataService {

  private ServerRepository serverRepository;

  public ServerDataService(ServerRepository serverRepository) {
    this.serverRepository = serverRepository;
  }

  @Transactional
  public Server saveServer(Server server) {
    Server newServer = serverRepository.findByUrl(server.getUrl());
    if (newServer != null) {
      String thisMethodName = new Object() {
      }.getClass().getEnclosingMethod().getName();
      String message = String.format("%s: Server with url %s already exist",
thisMethodName, server.getUrl());
      log.error(message);
      throw new ServerAlreadyExist(message);
    } else {
      server.setCounts(0);
      server.setServerId(randomUUID().toString());
      return serverRepository.save(server);
    }
  }

  public List<Server> getAllServers() {
    return serverRepository.findAll();
  }

  public Server getServerByUrl(String url) {
    return serverRepository.findByUrl(url);
  }

  @Transactional
  public Server updateServer(Server server) {
    Server newServer = serverRepository.findByUrl(server.getUrl());
    if (newServer == null) {
      String thisMethodName = new Object() {
      }.getClass().getEnclosingMethod().getName();
      String message = String.format("%s: Server with url %s not exist",
thisMethodName, server.getUrl());
      log.error(message);
      throw new ServerNotExist(message);
    } else {
      return serverRepository.saveAndFlush(newServer);
```

```
    }
  }

  public void inactiveServer(String url) {
    Server newServer = serverRepository.findByUrl(url);
    if (newServer == null) {
      String thisMethodName = new Object() {
      }.getClass().getEnclosingMethod().getName();
      String message = String.format("%s: Server with url %s not exist.",
thisMethodName, url);
      log.error(message);
      throw new ServerNotExist(message);
    } else {
      newServer.setActive(false);
      serverRepository.saveAndFlush(newServer);
    }
  }

  public void activeServer(String url) {
    Server newServer = serverRepository.findByUrl(url);
    if (newServer == null) {
      String thisMethodName = new Object() {
      }.getClass().getEnclosingMethod().getName();
      String message = String.format("%s: Server with url %s not exist.",
thisMethodName, url);
      log.error(message);
      throw new ServerNotExist(message);
    } else {
      newServer.setActive(true);
      serverRepository.saveAndFlush(newServer);
    }
  }

  public Server findServerWithMinCounts() {
    List<Server> newServer = serverRepository.findSrvWithMinCounts();
    if (newServer.isEmpty()) {
      String message = "Server not exist";
      throw new ServerNotExist(message);
    } else {
      return newServer.get(0);
    }
  }

  @Transactional
  public void incrementCountsOfServer(Server server) {
    server.setCounts(server.getCounts() + 1);
    serverRepository.saveAndFlush(server);
  }

  @Transactional
  public void decrementCountsOfServer(Server server) {
    server.setCounts(server.getCounts() - 1);
    serverRepository.saveAndFlush(server);
  }
}
```